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).

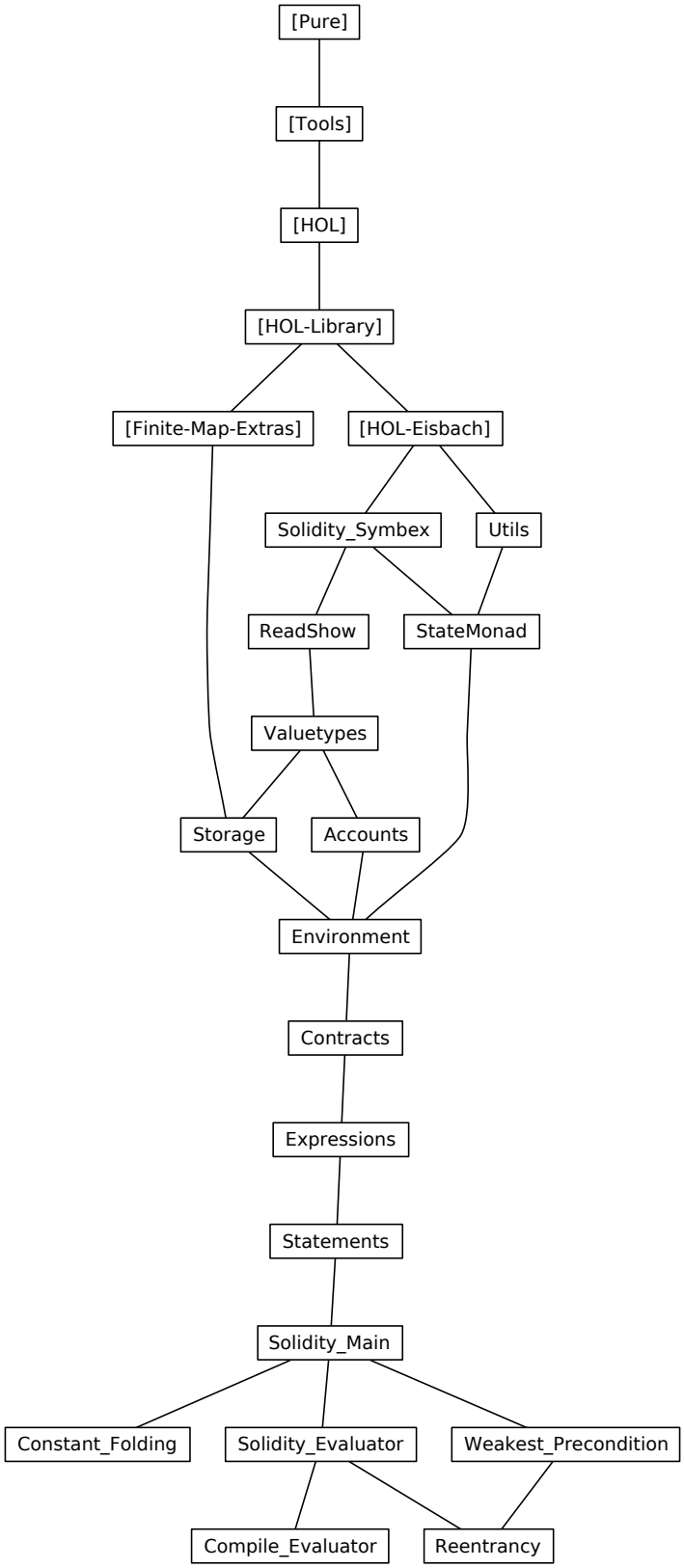

Figure 1.1: The Dependency Graph of the Isabelle Theories.

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)*›
  using *STR_is_bool_def* **by** *fastforce*

lemma *STR_is_bool_split:* ‹*STR_is_bool s* ⟹ *s = ''False'' ∨ s = ''True''*›
  by(*auto simp: STR_is_bool_def* $Read_{bool}$*_def* $Show_{bool}$*_def)*

lemma *Read_Show_bool_id:* ‹$Read_{bool}$ ($Show_{bool}$ *b) = b*›
  by(*auto simp:* $Read_{bool}$*_def* $Show_{bool}$*_def)*

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)*›
  using *strL_is_bool'_def* **by** *fastforce*

lemma *strL_is_bool'_split:* ‹*strL_is_bool' s* ⟹ *s = STR ''False'' ∨ s = STR ''True''*›
  by(*auto simp: strL_is_bool'_def* $ReadL_{bool}$*_def* $ShowL_{bool}$*_def)*

lemma *Read_Show_bool'_id[simp]:* ‹$ReadL_{bool}$ ($ShowL_{bool}$ *b) = b*›
  by(*auto simp:* $ReadL_{bool}$*_def* $ShowL_{bool}$*_def)*

lemma *true_neq_false[simp]: "ShowL$_{bool}$ True ≠ ShowL$_{bool}$ False"*
by *(metis Read_Show_bool'_id)*

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›
by*(simp add:nat_of_digit_def digit_of_nat_def)*

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''}›
by*(simp add:nat_of_digit_def digit_of_nat_def)*

```
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›
  by(code_simp, auto)

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 ≠ []›
  by (smt (verit, ccfv_threshold) nat_explode'.elims ord.lexordp_eq_simps(1) ord.lexordp_eq_simps(3))

lemma nat_explode_not_empty: ‹nat_explode n ≠ []›
  using nat_explode'_not_empty nat_explode_def by auto

lemma nat_explode'_ne_suc: ‹∃ n. nat_explode' (Suc n) ≠ nat_explode' n›
  by(rule exI [of _ ‹0::nat›], simp)

lemma nat_explode'_digit: ‹hd (nat_explode' n ) < 10›
proof(induct  ‹n›)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then show ?case proof (cases ‹n < 9›)
    case True
    then show ?thesis by simp
  next
    case False
    then show ?thesis
      by simp
  qed
qed

lemma div_ten_less: ‹n ≠ 0 ⟹ ((n::nat) div 10) < n›
  by simp

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))›
  by simp

lemma nat_explode_mod_10_ident: ‹map (λ x. x mod 10) (nat_explode' n) = nat_explode' n›
proof (cases ‹n < 10›)
  case True
  then show ?thesis by simp
next
  case False
  then show ?thesis
  proof (induct ‹n› rule: nat_less_induct)
  case (1 n) note * = this
```

```
  then show ?case
    using div_ten_less apply(simp (no_asm))
    using unroll_nat_explode'[of n] *
    by (smt (verit) list.simps(8) list.simps(9) mod_div_trivial mod_eq_self_iff_div_eq_0
              nat_explode'.simps zero_less_numeral)
  qed
qed

lemma nat_explode'_digits:
  ‹∀d ∈ set (nat_explode' n). d < 10›
proof
  fix d
  assume ‹d ∈ set (nat_explode' n)›
  then have ‹d ∈ set (map (λm. m mod 10) (nat_explode' n))›
    by (simp only: nat_explode_mod_10_ident)
  then show ‹d < 10›
    by auto
qed

lemma nat_explode_digits:
  ‹∀d ∈ set (nat_explode n). d < 10›
  using nat_explode'_digits [of n] by (simp only: nat_explode_def set_rev)

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 )›
  by(simp add:nat_implode_def)

lemma enumerate_suc: ‹enumerate (Suc n) l = map (λ (a,b). (a+1::nat,b)) (enumerate n l)›
proof (induction ‹l›)
  case Nil
  then show ?case by simp
next
  case (Cons a x) note * = this
  then show ?case apply(simp only:enumerate_simps)

    apply(simp only:‹enumerate (Suc n) x = map (λa. case a of (a, b) ⇒ (a + 1, b)) (enumerate n
x)› [symmetric])
    apply(simp)
    using *
    by (metis apfst_conv cond_case_prod_eta enumerate_Suc_eq)
qed

lemma mult_assoc_aux1:
 ‹(λ(p, y). 10 ^ p * y) ∘ (λ(a, y). (Suc a, y)) = (λ(p, y). (10::nat) * (10 ^ p) * y)›
  by(auto simp:o_def)

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(induct ‹l›)
  case Nil
  then show ?case by(simp)
next
  case (Cons a l)
  then show ?case  by(simp)
qed

lemma mult_assoc_aux2: ‹(λ(p, y). 10 * 10 ^ p * (y::nat)) = (λ(p, y). 10 * (10 ^ p * y))›
  by(auto)
```

```
lemma nat_implode_explode_id: ‹nat_implode (nat_explode n) = n›
proof (cases ‹n=0›)
  case True note * = this
  then show ?thesis
    by (simp add: nat_explode_def nat_implode_def)
next
  case False
  then show ?thesis
  proof (induct ‹n› rule: nat_less_induct)
    case (1 n) note * = this
    then  have
      **: ‹nat_implode (nat_explode (n div 10)) = n div 10›
    proof (cases ‹n div 10 = 0›)
      case True
      then show ?thesis by(simp add: nat_explode_def nat_implode_def)
    next
      case False
      then show ?thesis
      using div_ten_less[of ‹n›] *
      by(simp)
  qed
    then show ?case
    proof (cases ‹n < 10›)
      case True
      then show ?thesis  by(simp add: nat_explode_def nat_implode_def)
    next
      case False note *** = this
      then show ?thesis
        apply(simp (no_asm)  add:nat_explode_def del:rev_rev_ident)
        apply(simp only: bool.case(2))
        apply(simp del:nat_explode'.simps rev_rev_ident)
        apply(fold nat_explode_def)
        apply(simp only:nat_implode_append)
        apply(simp add:enumerate_suc)
        apply(simp only:mult_assoc_aux1)
        using mult_assoc_aux2 apply(simp)
        using fold_map_transfer[of ‹λ(p, y). 10 ^ p * y›
                                ‹(enumerate 0 (rev (nat_explode (n div 10))))›, simplified]
        apply(simp) apply(fold nat_implode_def) using **
        by simp
      qed
  qed
qed

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''}›
  unfolding Show_nat_def
  using  nat_explode_digits[of n]  img_digit_of_nat imageE image_set subsetI
  by (smt (verit) imageE image_set subsetI)

lemma Show_nat_not_empty: ‹(Show_nat n) ≠ []›
  by (simp add: Show_nat_def nat_explode_not_empty)

lemma not_hd: ‹L ≠ [] ⟹ e ∉ set(L) ⟹ hd L ≠ e›
  by auto

lemma Show_nat_not_neg'': ‹hd (Show_nat n) ≠ (CHR ''-'')›
  using Show_nat_not_neg[of ‹n›]
  Show_nat_not_empty[of ‹n›] not_hd[of ‹Show_nat n›]
  by auto

lemma Show_Read_nat_id: ‹STR_is_nat s ⟹ (Show_nat (Read_nat s) = s)›
  by(simp add:STR_is_nat_def)

lemma bar': ‹∀ d ∈ set l . d < 10 ⟹ map nat_of_digit (map digit_of_nat l) = l›
  using  nat_of_digit_digit_of_nat_id
  by (simp add: map_idI)

lemma Read_Show_nat_id: ‹Read_nat(Show_nat n) = n›
  apply(unfold Read_nat_def Show_nat_def)
  using bar' nat_of_digit_digit_of_nat_id nat_explode_digits
  using nat_implode_explode_id
  by presburger

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)›
  by(simp add:strL_is_nat'_def)

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›
  by auto

lemma Show_nat_ascii: ‹map String.ascii_of (Show_nat n) = Show_nat n›
  using Show_nat_not_neg digits_are_ascii
  by (meson map_idI subsetD)
```

```
lemma Read_Show_nat'_id: ‹ReadL_nat(ShowL_nat n) = n›
  apply(unfold ReadL_nat_def ShowL_nat_def, simp)
  by (simp add: Show_nat_ascii  Read_Show_nat_id)
```

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)›
  by(simp add:STR_is_int_def)

lemma Read_Show_id: ‹Read_int(Show_int(x)) = x›
  apply(code_simp)
  apply(auto simp:Show_nat_not_neg Read_Show_nat_id)[1]
  apply(thin_tac ‹¬ x < 0 ›)
  using Show_nat_not_neg''
  by blast

lemma STR_is_int_Show: ‹STR_is_int (Show_int n)›
  by(simp add:STR_is_int_def Read_Show_id)

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)›
```

```
  by(simp add:strL_is_int'_def)

lemma Read_ShowL_id: ‹ReadL_int (ShowL_int x) = x›
proof(cases ‹x < 0›)
  case True
  then show ?thesis using ShowL_int_def ReadL_int_def Show_int_def Show_nat_ascii
    by (metis (no_types, lifting) Read_Show_id String.ascii_of_Char comp_def implode.rep_eq
list.simps(9))
next
  case False
  then show ?thesis using ShowL_int_def ReadL_int_def Show_int_def Show_nat_ascii
    by (metis Read_Show_id String.explode_implode_eq comp_apply)
qed

lemma STR_is_int_ShowL: ‹strL_is_int' (ShowL_int n)›
  by(simp add:strL_is_int'_def Read_ShowL_id)

lemma String_Cancel: "a + (c::String.literal) = b + c ⟹ a = b"
using String.plus_literal.rep_eq
by (metis append_same_eq literal.explode_inject)

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 (cases x)
    case (Normal n)
    then show ?thesis
      by (metis n prod.swap_def swap_swap)
  next
    case (Exception e)
    then show ?thesis ..
```

qed

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" by auto
```

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"
  by auto
```

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

```
lemma bind_return [simp]: "(m ≫= return) = m"
proof -
  have "∀s. bind m return s = m s"
  proof
    fix s
    show "bind m return s = m s"
    proof (cases "m s" rule: result_cases)
      case (n a s)
      then show ?thesis by auto
    next
      case (e e)
      then show ?thesis by auto
    qed
  qed
  thus ?thesis by auto
qed
```

(≫=) 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
  fix s
  show "bind (bind m f) g s = bind m (λx. bind (f x) g) s"
    by (cases "m s" rule: result_cases, simp+)
qed
```

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"
  apply(insert assms, cases "m1 s")
```

```
  apply (metis StateMonad.bind.simps old.prod.case result.simps(5))
  by (metis bind.elims result.simps(6))


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"
  by (metis assms(1) assms(2) old.nat.exhaust old.nat.simps(4) old.nat.simps(5))

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"
  using assms(1) assms(2) assms(3) by auto

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"
  by (simp add: assms(1) assms(2) prod.case_eq_if)

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"
  by (simp add: assms(1) assms(2))

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"
  by (simp add: assms(1) assms(2) assms(3) option.case_eq_if)

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"
  using assms(1) assms(2) assms(3) by auto
```

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)))" **..**

lemma *"do {*
x ← return 1;
return 2;
return x
} = return 1"
by *auto*

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⦄"*
using *assms* **by** *(simp add: valid_def)*

lemma *strengthen:*
assumes *"⦃P⦄ f ⦃Q⦄, ⦃E⦄"*
and *"∀a s. Q a s ⟶ R a s"*

```
    shows "⦃P⦄ f ⦃R⦄, ⦃E⦄"
  unfolding valid_def
proof(rule allI[OF impI])
  fix s
  assume "P s"
  show "case f s of Normal (r, s') ⇒ R r s' | Exception e ⇒ E e"
  proof (cases "f s")
    case (n a s')
    then show ?thesis using assms valid_def ‹P s›
      by (metis case_prod_conv result.simps(5))
  next
    case (e e)
    then show ?thesis using assms valid_def ‹P s›
      by (metis result.simps(6))
  qed
qed

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⦄"
  unfolding valid_def by (metis assms wp_def)

lemma valid_wp: assumes "⦃P⦄ f ⦃Q⦄,⦃E⦄" shows "⋀s. P s ⟹ (wp f Q E s)"
  by (metis assms valid_def wp_def)

lemma put: "⦃λs. P () x⦄ put x ⦃P⦄,⦃E⦄"
  using valid_def by fastforce

lemma put':
  assumes "∀s. P s ⟶ Q () x"
  shows "⦃λs. P s⦄ put x ⦃Q⦄,⦃E⦄"
  using assms weaken[OF put, of P Q x E] by blast

lemma wpput[wprule]:
  assumes "P () x"
  shows "wp (put x) P E s"
  unfolding wp_def using assms by simp

lemma get: "⦃λs. P s s⦄ get ⦃P⦄,⦃E⦄"
  using valid_def by fastforce

lemma get':
  assumes "∀s. P s ⟶ Q s s"
  shows "⦃λs. P s⦄ get ⦃Q⦄,⦃E⦄"
  using assms weaken[OF get] by blast

lemma wpget[wprule]:
  assumes "P s s"
  shows "wp get P E s"
  unfolding wp_def using assms by simp

lemma return: "⦃λs. P x s⦄ return x ⦃P⦄,⦃E⦄"
  using valid_def by fastforce

lemma return':
  assumes "∀s. P s ⟶ Q x s"
  shows "⦃λs. P s⦄ return x ⦃Q⦄,⦃E⦄"
  using assms weaken[OF return, of P Q x E] by blast
```

```
lemma wpreturn[wprule]:
  assumes "P x s"
  shows "wp (return x) P E s"
  unfolding wp_def using assms by simp

lemma bind:
  assumes "∀x. ⦃B x⦄ g x ⦃C⦄,⦃E⦄"
      and "⦃A⦄ f ⦃B⦄,⦃E⦄"
    shows "⦃A⦄ f ⤜ g ⦃C⦄,⦃E⦄"
  unfolding valid_def
proof (rule allI[OF impI])
  fix s assume a: "A s"
  show "case (f ⤜ g) s of Normal (r, s') ⇒ C r s' | Exception e ⇒ E e"
  proof (cases "f s" rule:result_cases)
    case nf: (n a s')
    with assms(2) a have b: "B a s'" using valid_def[where ?f=f] by auto
    then show ?thesis
    proof (cases "g a s'" rule:result_cases)
      case ng: (n a' s'')
      with assms(1) b have c: "C a' s''" using valid_def[where ?f="g a"] by fastforce
      moreover from ng nf have "(f ⤜ g) s = Normal (a', s'')" by simp
      ultimately show ?thesis by simp
    next
      case eg: (e e)
      with assms(1) b have c: "E e" using valid_def[where ?f="g a"] by fastforce
      moreover from eg nf have "(f ⤜ g) s = Exception e" by simp
      ultimately show ?thesis by simp
    qed
  next
    case (e e)
    with a assms(2) have "E e" using valid_def[where ?f=f] by auto
    moreover from e have "(f ⤜ g) s = Exception e" by simp
    ultimately show ?thesis by simp
  qed
qed

lemma wpbind[wprule]:
  assumes "wp f (λa. (wp (g a) P E)) E s"
  shows "wp (f ⤜ g) P E s"
proof (cases "f s" rule:result_cases)
  case nf: (n a s')
  then have **:"wp (g a) P E s'" using wp_def[of f "λa. wp (g a) P E"] assms by simp
  show ?thesis
  proof (cases "g a s'" rule:result_cases)
    case ng: (n a' s'')
    then have "P a' s''" using wp_def[of "g a" P] ** by simp
    moreover from nf ng have "(f ⤜ g) s = Normal (a', s'')" by simp
    ultimately show ?thesis using wp_def by fastforce
  next
    case (e e)
    then have "E e" using wp_def[of "g a" P] ** by simp
    moreover from nf e have "(f ⤜ g) s = Exception e" by simp
    ultimately show ?thesis using wp_def by fastforce
  qed
next
  case (e e)
  then have "E e" using wp_def[of f "λa. wp (g a) P E"] assms by simp
  moreover from e have "(f ⤜ g) s = Exception e" by simp
  ultimately show ?thesis using wp_def by fastforce
qed

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"
  using assms unfolding wp_def by simp

lemma throw:
  assumes "E x"
  shows "⦃P⦄ throw x ⦃Q⦄, ⦃E⦄"
  using valid_def assms by fastforce

lemma wpthrow[wprule]:
  assumes "E x"
  shows "wp (throw x) P E s"
  unfolding wp_def using assms by simp

lemma applyf:
    "⦃λs. P (f s) s⦄ applyf f ⦃λa s. P a s⦄,⦃E⦄"
  by (simp add: valid_def)

lemma applyf':
  assumes "∀s. P s ⟶ Q (f s) s"
  shows "⦃λs. P s⦄ applyf f ⦃λa s. Q a s⦄,⦃E⦄"
  using assms weaken[OF applyf] by blast

lemma wpapplyf[wprule]:
  assumes "P (f s) s"
  shows "wp (applyf f) P E s"
  unfolding wp_def using assms by simp

lemma modify:
  "⦃λs. P () (f s)⦄ modify f ⦃P⦄, ⦃E⦄"
  apply simp
  apply (rule bind,rule allI)
  apply (rule put)
  apply (rule get)
  done

lemma modify':
  assumes "∀s. P s ⟶ Q () (f s)"
  shows "⦃λs. P s⦄ modify f ⦃Q⦄, ⦃E⦄"
  using assms weaken[OF modify, of P Q _ E] by blast

lemma wpmodify[wprule]:
  assumes "P () (f s)"
  shows "wp (modify f) P E s"
  unfolding wp_def using assms by simp

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"
  by (metis assms(1) assms(2) not0_implies_Suc old.nat.simps(4) old.nat.simps(5))

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"
  using assms by simp

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"
  using assms unfolding wp_def by (simp split: option.split)

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"
  using assms unfolding wp_def by (auto split:option.split)

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"
  using assms unfolding wp_def
  by (simp split: prod.split)

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⦈}"
  by (wpvcg, simp)
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 -
  have a1: "⋀P. (∀b::nat. P b) ⟹ (∀b>0. P ((b-1)::nat))" by simp
  have b2: "∀b::nat. (∀(c::int)<2^b. (c + 2^b) < 2^(Suc b))" by simp
  show ?thesis using a1[OF b2] assms by simp
qed

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 -
  have a1: "⋀P. (∀b::nat. P b) ⟹ (∀b>0. P ((b-1)::nat))" by simp
  have b2: "∀b::nat. (∀(c::int)<2^(Suc b). c≥0 ⟶ (c - 2^b) < 2^b)" by simp
  show ?thesis using a1[OF b2] assms by simp
qed

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)"
  using upper_bound2 assms by auto

lemma lower_bound:
    fixes b::nat
  assumes "b>0"
    shows "∀(c::int) ≥ -(2^(b-1)). (-c + 2^(b-1) - 1 < 2^b)"
proof -
  have a1: "⋀P. (∀b::nat. P b) ⟹ (∀b>0. P ((b-1)::nat))" by simp
  have b2: "∀b::nat. ∀(c::int) ≥ -(2^b). (-c + (2^b) - 1) < 2^(Suc b)" by simp
  show ?thesis using a1[OF b2] assms by simp
qed

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))"
  using upper_bound2 assms by auto

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 -
  from assms have "v + 2^(b-1) ≥ 0" by simp
  moreover from assms have "v + (2^(b-1)) < 2^b" using upper_bound[of b] by auto
  ultimately have "(v + 2^(b-1)) mod (2^b) = v + 2^(b-1)" by simp
  moreover from assms have "createSInt b v=ShowL_int (-(2^(b-1)) + (v+2^(b-1)) mod (2^b))" unfolding
createSInt_def by simp
  ultimately show ?thesis by simp
qed

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 -
  from assms have "-v + 2^(b-1) - 1 ≥ 0" by simp
  moreover from assms have "-v + 2^(b-1) - 1 < 2^b" using lower_bound[of b] by auto
  ultimately have "(-v + 2^(b-1) - 1) mod (2^b) = (-v + 2^(b-1) - 1)" by simp
  moreover from assms have "createSInt b v= ShowL_int (2^(b-1) - (-v+2^(b-1)-1) mod (2^b) - 1)" un-
folding createSInt_def by simp
```

```
  ultimately show ?thesis by simp
qed

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" using createSInt_id_g0 createSInt_id_l0 assms unfolding
createSInt_def by simp


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"
unfolding createUInt_def by (simp add: assms(1) assms(2))

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"
    by (metis Types.exhaust convert.simps(1) convert.simps(2) convert.simps(4) convert.simps(5)
order_refl)


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"
  using assms by (cases x, auto)

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"
using assms unfolding addBalance_def by (simp add:Let_def split:if_split_asm)

lemma addBalance_val2:
  assumes "addBalance ad val acc = Some acc'"
    shows "ReadL_int (bal (acc ad)) + ReadL_int val < 2^256"
using assms unfolding addBalance_def by (simp add:Let_def split:if_split_asm)

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
  fix ad'
  show "ReadL_int (bal (acc' ad')) ≥ 0 ∧ ReadL_int (bal (acc' ad')) < 2 ^ 256"
  proof (cases "ReadL_int val ≥ 0")
    case t1: True
    define v where v_def: "v = ReadL_int (bal (acc ad)) + ReadL_int val"
```

```
    with assms(2) have "v ≥0" by (simp add: t1)
    then show ?thesis
    proof (cases "v < 2^256")
      case t2: True
      with t1 v_def have "addBalance ad val acc = Some (acc(ad := acc ad⦇bal:=ShowL_int v⦈))" unfolding
addBalance_def by simp
      with t2 ‘v ≥0‘ show ?thesis using assms Read_ShowL_id by auto
    next
      case False
      with t1 v_def show ?thesis using assms(1) unfolding addBalance_def by simp
    qed
  next
    case False
    then show ?thesis using assms(1) unfolding addBalance_def by simp
  qed
qed

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 -
  define v where "v = ReadL_int (bal (acc ad)) + ReadL_int val"
  with assms have "acc' = acc(ad := acc ad⦇bal:=ShowL_int v⦈)" unfolding addBalance_def by (simp
add:Let_def split:if_split_asm)
  thus ?thesis using v_def Read_ShowL_id assms by (simp split:if_split_asm)
qed

lemma addBalance_mono:
  assumes "addBalance ad val acc = Some acc'"
    shows "ReadL_int (bal (acc' ad)) ≥ ReadL_int (bal (acc ad))"
proof -
  define v where "v = ReadL_int (bal (acc ad)) + ReadL_int val"
  with assms have "acc' = acc(ad := acc ad⦇bal:=ShowL_int v⦈)" unfolding addBalance_def by (simp
add:Let_def split:if_split_asm)
  thus ?thesis using v_def Read_ShowL_id assms unfolding addBalance_def by (simp split:if_split_asm)
qed

lemma addBalance_eq:
  assumes "addBalance ad val acc = Some acc'"
      and "ad ≠ ad'"
    shows "bal (acc ad') = bal (acc' ad')"
proof -
  define v where "v = ReadL_int (bal (acc ad)) + ReadL_int val"
  with assms have "acc' = acc(ad := acc ad⦇bal:=ShowL_int v⦈)" unfolding addBalance_def by (simp
add:Let_def split:if_split_asm)
  thus ?thesis using v_def Read_ShowL_id assms by (simp split:if_split_asm)
qed

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"
using assms unfolding subBalance_def by (simp split:if_split_asm)
```

```
lemma subBalance_val2:
  assumes "subBalance ad val acc = Some acc'"
    shows "ReadL_int (bal (acc ad)) - ReadL_int val ≥ 0"
using assms unfolding subBalance_def by (simp split:if_split_asm)

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 -
  define v where "v = ReadL_int (bal (acc ad)) - ReadL_int val"
  with assms have "acc' = acc(ad := acc ad⦇bal:=ShowL_int v⦈)" unfolding subBalance_def by (simp
add:Let_def split:if_split_asm)
  thus ?thesis using v_def Read_ShowL_id assms by (simp split:if_split_asm)
qed

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
  fix ad'
  show "ReadL_int (bal (acc' ad')) ≥ 0 ∧ ReadL_int (bal (acc' ad')) < 2 ^ 256"
  proof (cases "ReadL_int val ≥ 0")
    case t1: True
    define v where v_def: "v = ReadL_int (bal (acc ad)) - ReadL_int val"
    with assms(2) t1 have "v < 2 ^ 256" by (smt (verit))
    then show ?thesis
    proof (cases "v ≥ 0")
      case t2: True
      with t1 v_def have "subBalance ad val acc = Some (acc(ad := acc ad⦇bal:=ShowL_int v⦈))" unfolding
subBalance_def by simp
      with t2 ‘v < 2 ^ 256‘ show ?thesis using assms Read_ShowL_id by auto
    next
      case False
      with t1 v_def show ?thesis using assms(1) unfolding subBalance_def by simp
    qed
  next
    case False
    then show ?thesis using assms(1) unfolding subBalance_def by simp
  qed
qed

lemma subBalance_mono:
  assumes "subBalance ad val acc = Some acc'"
    shows "ReadL_int (bal (acc ad)) ≥ ReadL_int (bal (acc' ad))"
proof -
  define v where "v = ReadL_int (bal (acc ad)) - ReadL_int val"
  with assms have "acc' = acc(ad := acc ad⦇bal:=ShowL_int v⦈)" unfolding subBalance_def by (simp
add:Let_def split:if_split_asm)
  thus ?thesis using v_def Read_ShowL_id assms unfolding subBalance_def by (simp split:if_split_asm)
qed

lemma subBalance_eq:
  assumes "subBalance ad val acc = Some acc'"
     and "ad ≠ ad'"
    shows "(bal (acc ad')) = (bal (acc' ad'))"
proof -
  define v where "v = ReadL_int (bal (acc ad)) - ReadL_int val"
  with assms have "acc' = acc(ad := acc ad⦇bal:=ShowL_int v⦈)" unfolding subBalance_def by (simp
add:Let_def split:if_split_asm)
  thus ?thesis using v_def Read_ShowL_id assms by (simp split:if_split_asm)
qed
```

```
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 -
  from assms(1) obtain acc''
    where *: "subBalance ads val acc = Some acc''"
      and **: "addBalance addr val acc'' = Some acc'" by (simp add: subBalance_def transfer_def
split:if_split_asm)
  then show "ReadL_int val ≥ 0" using subBalance_val1[OF *] by simp
qed

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 -
  from assms(1) obtain acc''
    where *: "subBalance ads val acc = Some acc''"
      and **: "addBalance addr val acc'' = Some acc'" by (simp add: subBalance_def transfer_def
split:if_split_asm)

  have "ReadL_int (bal (acc'' addr)) + ReadL_int val < 2^256" using addBalance_val2[OF **] by simp
  with * show ?thesis using assms(2) subBalance_eq[OF *] by simp
qed

lemma transfer_val3:
  assumes "transfer ads addr val acc = Some acc'"
    shows "ReadL_int (bal (acc ads)) - ReadL_int val ≥ 0"
using assms by (auto simp add: Let_def subBalance_def transfer_def split:if_split_asm)

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 -
  from assms(1) obtain acc''
    where *: "subBalance ads val acc = Some acc''"
      and **: "addBalance addr val acc'' = Some acc'" by (simp add: subBalance_def transfer_def
split:if_split_asm)

  with assms(2) have "ReadL_int (bal (acc addr)) = ReadL_int (bal (acc'' addr))" using subBalance_eq[OF
*] by simp
  moreover have "ReadL_int (bal (acc' addr)) = ReadL_int (bal (acc'' addr)) + ReadL_int val" using
addBalance_add[OF **] by simp
  ultimately show ?thesis using Read_ShowL_id by simp
qed

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 -
  from assms(1) obtain acc''
    where *: "subBalance ads val acc = Some acc''"
      and **: "addBalance addr val acc'' = Some acc'" by (simp add: subBalance_def transfer_def
```

```
split:if_split_asm)

  then have "ReadL_int (bal (acc'' ads)) = ReadL_int (bal (acc ads)) - ReadL_int val" using
subBalance_sub[OF *] by simp
  moreover from assms(2) have "ReadL_int (bal (acc' ads)) = ReadL_int (bal (acc'' ads))" using
addBalance_eq[OF **] by simp
  ultimately show ?thesis using Read_ShowL_id by simp
qed

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 -
  from assms obtain acc'' where *: "subBalance ad val acc = Some acc''" by (simp add: subBalance_def
transfer_def split:if_split_asm)
  with assms have **: "addBalance ad val acc'' = Some acc'" by (simp add: transfer_def
split:if_split_asm)
  moreover from * have "ReadL_int (bal (acc'' ad)) = ReadL_int (bal (acc ad)) - ReadL_int val" using
subBalance_sub by simp
  moreover from ** have "ReadL_int (bal (acc' ad)) = ReadL_int (bal (acc'' ad)) + ReadL_int val" using
addBalance_add by simp
  ultimately show ?thesis by simp
qed

lemma transfer_mono:
  assumes "transfer ads addr val acc = Some acc'"
  shows "ReadL_int (bal (acc' addr)) ≥ ReadL_int (bal (acc addr))"
proof -
  from assms(1) obtain acc''
    where *: "subBalance ads val acc = Some acc''"
      and **: "addBalance addr val acc'' = Some acc'" by (simp add: subBalance_def transfer_def
split:if_split_asm)

  show ?thesis
  proof (cases "addr = ads")
    case True
    with * have "acc'' = acc(addr:=acc addr⦇bal := ShowL_int (ReadL_int (bal (acc addr)) - ReadL_int
val)⦈)" by (simp add:Let_def subBalance_def split: if_split_asm)
    moreover from ** have "acc'=acc''(addr:=acc'' addr⦇bal := ShowL_int (ReadL_int (bal (acc'' addr)) +
ReadL_int val)⦈)" unfolding addBalance_def by (simp add:Let_def split: if_split_asm)
    ultimately show ?thesis using Read_ShowL_id by auto
  next
    case False
    then have "ReadL_int (bal (acc addr)) ≤ ReadL_int (bal (acc'' addr))" using subBalance_eq[OF *] by
simp
    also have "... ≤ ReadL_int (bal (acc' addr))" using addBalance_mono[OF **] by simp
    finally show ?thesis .
  qed
qed

lemma transfer_eq:
  assumes "transfer ads addr val acc = Some acc'"
      and "ad ≠ ads"
      and "ad ≠ addr"
    shows "bal (acc' ad) = bal (acc ad)"
using assms by (auto simp add: Let_def addBalance_def subBalance_def transfer_def split:if_split_asm)

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
  fix ad'
```

```
  from assms(1) obtain acc'' where "subBalance ads val acc = Some acc''" and "addBalance addr val
acc'' = Some acc'" by (simp add: subBalance_def transfer_def split: if_split_asm)
  with subBalance_limit[OF _ assms(2)]
  show "ReadL_int (bal (acc' ad')) ≥ 0 ∧ ReadL_int (bal (acc' ad')) < 2 ^ 256"
    using addBalance_limit by presburger
qed

lemma transfer_type_same:
  assumes "transfer ads addr val acc = Some acc'"
    shows "type (acc' ad) = type (acc ad)"
using assms by (auto simp add: Let_def addBalance_def subBalance_def transfer_def split:if_split_asm)

lemma transfer_contracts_same:
  assumes "transfer ads addr val acc = Some acc'"
    shows "contracts (acc' ad) = contracts (acc ad)"
using assms by (auto simp add: Let_def addBalance_def subBalance_def transfer_def split:if_split_asm)


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'')"* **by** *eval*

lemma *hash_explode:*
  "String.explode (hash l i) = String.explode i @ (String.explode (STR ''.'') @ String.explode l)"
  unfolding *hash_def* **by** *(simp add: plus_literal.rep_eq)*

lemma *hash_dot:*
  "String.explode (hash l i) ! length (String.explode i) = CHR ''.''"
  unfolding *hash_def* **by** *(simp add: Literal.rep_eq plus_literal.rep_eq)*

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 *(rule ccontr)*
  assume *"¬ (l = l' ∧ i = i')"*
  then consider *(1) "i≠i'" | (2) "i=i' ∧ l≠l'"* **by** *auto*
  then have *"String.explode (hash l i) ≠ String.explode (hash l' i')"*
  proof *cases*
    case *1*
    then have *neq: "(String.explode i) ≠ (String.explode i')"* **by** *(metis literal.explode_inverse)*

    consider *(1) "length (String.explode i) = length (String.explode i')" | (2) "length (String.explode i) < length (String.explode i')" | (3) "length (String.explode i) > length (String.explode i')"* **by** *linarith*
    then show *?thesis*
    proof *(cases)*
      case *1*
      then obtain *j* **where** *"String.explode i ! j ≠ String.explode i' ! j"* **using** *neq nth_equalityI* **by** *auto*
      then show *?thesis* **using** *1 plus_literal.rep_eq* **unfolding** *hash_def* **by** *force*
    next
      case *2*
      then have *"String.explode i' ! length (String.explode i) ≠ CHR ''.''"* **using** *assms(3)* **by** *(metis nth_mem)*
      then have *"String.explode (hash l' i') ! length (String.explode i) ≠ CHR ''.''"* **using** *2 hash_explode[of l' i']* **by** *(simp add: nth_append)*

```
      moreover have "String.explode (hash l i) ! length (String.explode i) = CHR ''.''" using hash_dot
by simp
      ultimately show ?thesis by auto
    next
      case 3
      then have "String.explode i ! length (String.explode i') ≠ CHR ''.''" using assms(2) by (metis
nth_mem)
      then have "String.explode (hash l i) ! length (String.explode i') ≠ CHR ''.''" using 3
hash_explode[of l i] by (simp add: nth_append)
      moreover have "String.explode (hash l' i') ! length (String.explode i') = CHR ''.''" using
hash_dot by simp
      ultimately show ?thesis by auto
    qed
  next
    case 2
    then show ?thesis using hash_explode literal.explode_inject by force
  qed
  then show False using assms(1) by simp
qed
```

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"
  by eval
```

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|)" by eval
```

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"
  by eval

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" by eval
```

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"
  by eval
```

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"
  by eval

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"
  unfolding emptyEnv_def by simp

lemma emptyEnv_members[simp]:
  "contract (emptyEnv a c s v) = c"
  unfolding emptyEnv_def by simp
```

```
lemma emptyEnv_sender[simp]:
  "sender (emptyEnv a c s v) = s"
  unfolding emptyEnv_def by simp

lemma emptyEnv_svalue[simp]:
  "svalue (emptyEnv a c s v) = v"
  unfolding emptyEnv_def by simp

lemma emptyEnv_denvalue[simp]:
  "denvalue (emptyEnv a c s v) = {$$}"
  unfolding emptyEnv_def by simp

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'"
by (auto split:option.split_asm)

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"
  by (auto split:option.split)

lemma updateEnvDup_sender[simp]: "sender (updateEnvDup i t v e) = sender e"
  by (auto split:option.split)

lemma updateEnvDup_svalue[simp]: "svalue (updateEnvDup i t v e) = svalue e"
  by (auto split:option.split)

lemma updateEnvDup_dup:
  assumes "i≠i'" shows "fmlookup (denvalue (updateEnvDup i t v e)) i' = fmlookup (denvalue e) i'"
proof (cases "fmlookup (denvalue e) i = None")
  case True
  then show ?thesis using assms by simp
next
  case False
  then obtain e' where "fmlookup (denvalue e) i = Some e'" by auto
  then show ?thesis using assms by simp
qed

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 -
  have "address (updateEnv x t1 v1 (updateEnv y t2 v2 e)) = address (updateEnv y t2 v2 (updateEnv x t1
v1 e))" by simp
  moreover from assms have "denvalue (updateEnv x t1 v1 (updateEnv y t2 v2 e)) = denvalue (updateEnv
y t2 v2 (updateEnv x t1 v1 e))" using Finite_Map.fmupd_reorder_neq[of x y "(t1,v1)" "(t2,v2)"] by simp
  ultimately show ?thesis by simp
qed
```

```
lemma uEO_in:
  assumes "i |∈| fmdom (denvalue e)"
  shows "updateEnvOption i t v e = None"
  using assms by auto

lemma uEO_n_In:
  assumes "¬ i |∈| fmdom (denvalue e)"
  shows "updateEnvOption i t v e = Some (updateEnv i t v e)"
  using assms by auto

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)" by solidity_symbex
lemma "astack (STR ''id2'') (Value TBool) (KValue (STR ''False'')) (mystack, myenv) =
(mystack2,myenv2)" by solidity_symbex
```

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)"*
using *assms*
proof *(cases rule:decl.elims)*
case *(1 t uu uv uw ux c m k env)*
then show *?thesis* **by** *simp*
next
case *(2 t v t' uy uz va vb c m k e)*
show *?thesis*
proof *(cases "convert t' t v")*
case *None*
with *2* **show** *?thesis* **by** *simp*
next
case *s: (Some a)*
with *2 s* **show** *?thesis* **by** *simp*
qed
next
case *(3 vd ve vb vf vg vh vi vj)*
then show *?thesis* **by** *simp*
next
case *(4 vd ve vb vf vg vh vi vj)*
then show *?thesis* **by** *simp*

```
next
  case (5 vd ve vb vf vg vh vi vj)
  then show ?thesis by simp
next
  case (6 vd va ve vf vg vh vi vj)
  then show ?thesis by simp
next
  case (7 vd va ve vf vg vh vi vj)
  then show ?thesis by simp
next
  case (8 vd va ve vf vg vh vi vj)
  then show ?thesis by simp
next
  case (9 x t p vk cd vl vm c m k env)
  define l where "l = ShowL_nat (toploc c)"
  obtain c' where c_def: "∃x. (x, c') = allocate c" unfolding allocate_def by simp
  show ?thesis
  proof (cases "cpm2m p l x t cd c'")
    case None
    with 9 l_def c_def show ?thesis unfolding allocate_def by simp
  next
    case s2: (Some a)
    with 9 l_def c_def show ?thesis unfolding allocate_def by simp
  qed
next
  case (10 x t p vn vo mem vp c m k env)
  define l where "l = ShowL_nat (toploc c)"
  obtain c' where c_def: "∃x. (x, c') = allocate c" unfolding allocate_def by simp
  show ?thesis
  proof (cases "cpm2m p l x t mem c'")
    case None
    with 10 l_def c_def show ?thesis unfolding allocate_def by simp
  next
    case s2: (Some a)
    with 10 l_def c_def show ?thesis unfolding allocate_def by simp
  qed
next
  case (11 v vr vs vt vu vv vw)
  then show ?thesis by simp
next
  case (12 vq vs vt vu vv vw)
  then show ?thesis by simp
next
  case (13 vq vc vb vs vt vu vv vw)
  then show ?thesis by simp
next
  case (14 v vc vb vs vt vu vv vw)
  then show ?thesis by simp
next
  case (15 vq vc vb vt vu vv vw)
  then show ?thesis by simp
next
  case (16 vq vc vb vs vt vu vv vw)
  then show ?thesis by simp
next
  case (17 vq vr vt vu vv vw)
  then show ?thesis by simp
next
  case (18 x t vx vy vz wa c m k env)
  then show ?thesis by simp
next
  case (19 x t p wb wc mem wd c m k env)
  show ?thesis
  proof (cases cp)
```

```
    case True
    define l where "l = ShowL_nat (toploc m)"
    obtain m' where m'_def: "∃x. (x, m') = allocate m" unfolding allocate_def by simp
    then show ?thesis
    proof (cases "cpm2m p l x t mem m'")
      case None
      with 19 l_def m'_def show ?thesis unfolding allocate_def by simp
    next
      case s2: (Some a)
      with 19 l_def m'_def show ?thesis unfolding allocate_def by simp
    qed
  next
    case False
    with 19 show ?thesis by simp
  qed
next
  case (20 x t p we wf wg wh c m k env)
  then show ?thesis by simp
next
  case (21 x t p wi wj cd wk wl c m k env)
  define l where "l = ShowL_nat (toploc m)"
  obtain m' where m'_def: "∃x. (x, m') = allocate m" unfolding allocate_def by simp
  then show ?thesis
  proof (cases "cpm2m p l x t cd m'")
    case None
    with 21 l_def m'_def show ?thesis unfolding allocate_def by simp
  next
    case s2: (Some a)
    with 21 l_def m'_def show ?thesis unfolding allocate_def by simp
  qed
next
  case (22 x t p x' t' wm wn wo s c m k env)
  define l where "l = ShowL_nat (toploc m)"
  obtain m' where m'_def: "∃x. (x, m') = allocate m" unfolding allocate_def by simp
  show ?thesis
  proof (cases "cps2m p l x' t' (s (address env)) m'")
    case None
    with 22 l_def m'_def show ?thesis unfolding allocate_def by simp
  next
    case s2: (Some a)
    with 22 l_def m'_def show ?thesis unfolding allocate_def by simp
  qed
next
  case (23 v wr ws wt wu wv ww)
  then show ?thesis by simp
next
  case (24 va v ws wt wu wv ww)
  then show ?thesis by simp
next
  case (25 wq vc vb ws wt wu wv ww)
  then show ?thesis by simp
next
  case (26 v vc vb ws wt wu wv ww)
  then show ?thesis by simp
next
  case (27 v vc vb ws wt wu wv ww)
  then show ?thesis by simp
next
  case (28 wq vc v ws wt wu wv ww)
  then show ?thesis by simp
next
  case (29 wq vc v ws wt wu wv ww)
  then show ?thesis by simp
next
```

```
  case (30 wq vc v ws wt wu wv ww)
  then show ?thesis by simp
next
  case (31 wq vc va vd ws wt wu wv ww)
  then show ?thesis by simp
next
  case (32 wq vc va ws wt wu wv ww)
  then show ?thesis by simp
next
  case (33 va v wt wu wv ww)
  then show ?thesis by simp
next
  case (34 wq vc vb wt wu wv ww)
  then show ?thesis by simp
next
  case (35 v vc vb wt wu wv ww)
  then show ?thesis by simp
next
  case (36 v vc vb wt wu wv ww)
  then show ?thesis by simp
next
  case (37 wq vc v wt wu wv ww)
  then show ?thesis by simp
next
  case (38 wq vc v wt wu wv ww)
  then show ?thesis by simp
next
  case (39 wq vc v wt wu wv ww)
  then show ?thesis by simp
next
  case (40 wq vc va vd wt wu wv ww)
  then show ?thesis by simp
next
  case (41 wq vc va wt wu wv ww)
  then show ?thesis by simp
next
  case (42 x t p wx wy wz xa xb c m k env)
  then show ?thesis by simp
next
  case (43 t t' p xc xd xe xf xg c m k e)
  then show ?thesis by simp
next
  case (44 v va xk xl xm xn xo)
  then show ?thesis by simp
next
  case (45 v va ve vd xk xl xm xn xo)
  then show ?thesis by simp
next
  case (46 v va ve vd xk xl xm xn xo)
  then show ?thesis by simp
next
  case (47 v va ve vd xk xl xm xn xo)
  then show ?thesis by simp
next
  case (48 v xj xk xl xm xn xo)
  then show ?thesis by simp
next
  case (49 xi xk xl xm xn xo)
  then show ?thesis by simp
next
  case (50 xi vc vb xk xl xm xn xo)
  then show ?thesis by simp
next
  case (51 xi vc vb xk xl xm xn xo)
```

```
  then show ?thesis by simp
next
  case (52 xi vc vb xk xl xm xn xo)
  then show ?thesis by simp
qed

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 -
  consider (b1) "max b1 b2 = b1" | (b2) "max b1 b2 = b2" by (metis max_def)
  then show ?thesis
```

```
  proof cases
    case b1
    then show ?thesis using assms(1) by simp
  next
    case b2
    then show ?thesis using assms(2) by simp
  qed
qed

lemma vbits_ge_0: "(x::nat)∈vbits ⟹ x>0" by auto
```

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" using assms by simp

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" using assms by simp

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')"
  using incrementAccountContracts_def by simp

lemma incrementAccountContracts_bal[simp]:
  "bal (accounts (incrementAccountContracts ad st) ad') = bal (accounts st ad')"
  using incrementAccountContracts_def by simp

lemma incrementAccountContracts_stack[simp]:
  "stack (incrementAccountContracts ad st) = stack st"
  using incrementAccountContracts_def by simp

lemma incrementAccountContracts_memory[simp]:
  "memory (incrementAccountContracts ad st) = memory st"
  using incrementAccountContracts_def by simp

lemma incrementAccountContracts_storage[simp]:
  "storage (incrementAccountContracts ad st) = storage st"
  using incrementAccountContracts_def by simp

lemma incrementAccountContracts_gas[simp]:
  "gas (incrementAccountContracts ad st) = gas st"
  using incrementAccountContracts_def by simp

lemma gas_induct:
  assumes "⋀s. ∀s'. gas s' < gas s ⟶ P s' ⟹ P s"
```

```
  shows "P s" using assms by (rule Nat.measure_induct[of "λs. gas s"])

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"
```

```
  using assms init_def by simp

lemma init_s12[simp]:
  assumes "i |∈| fmdom (denvalue e)"
  shows "init ct i e = e"
proof (cases "fmlookup ct i")
  case None
  then show ?thesis using init_def by simp
next
  case (Some a)
  then show ?thesis
  proof (cases a)
    case (Method _)
    with Some show ?thesis using init_def by simp
  next
    case (Function _)
    with Some show ?thesis using init_def by simp
  next
    case (Var tp)
    with Some have "init ct i e = updateEnvDup i (Storage tp) (Storeloc i) e" using init_def by simp
    moreover from assms have "updateEnvDup i (Storage tp) (Storeloc i) e = e" by auto
    ultimately show ?thesis by simp
  qed
qed

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"
  using assms init_def by auto

lemma init_s21[simp]:
  assumes "fmlookup ct i = None"
  shows "init ct i e = e"
  using assms init_def by auto

lemma init_s22[simp]:
  assumes "fmlookup ct i = Some (Method m)"
  shows "init ct i e = e"
  using assms init_def by auto

lemma init_s23[simp]:
  assumes "fmlookup ct i = Some (Function f)"
  shows "init ct i e = e"
  using assms init_def by auto

lemma init_commte: "comp_fun_commute (init ct)"
proof
  fix x y
  show "init ct y ∘ init ct x = init ct x ∘ init ct y"
  proof
    fix e
    show "(init ct y ∘ init ct x) e = (init ct x ∘ init ct y) e"
    proof (cases "fmlookup ct x")
      case None
      then show ?thesis by simp
    next
      case s1: (Some a)
      then show ?thesis
      proof (cases a)
        case (Method _)
        with s1 show ?thesis by simp
      next
        case (Function _)
```

```
  with s1 show ?thesis by simp
next
  case v1: (Var tp)
  then show ?thesis
  proof (cases "x |∈| fmdom (denvalue e)")
    case True
    with s1 v1 have *: "init ct x e = e" by auto
    then show ?thesis
    proof (cases "fmlookup ct y")
      case None
      then show ?thesis by simp
    next
      case s2: (Some a)
      then show ?thesis
      proof (cases a)
        case (Method _)
        with s2 show ?thesis by simp
      next
        case (Function _)
        with s2 show ?thesis by simp
      next
        case v2: (Var tp')
        then show ?thesis
        proof (cases "y |∈| fmdom (denvalue e)")
          case t1: True
          with s1 v1 True s2 v2 show ?thesis by fastforce
        next
          define e' where "e' = updateEnv y (Storage tp') (Storeloc y) e"
          case False
          with s2 v2 have "init ct y e = e'" using e'_def by auto
          with s1 v1 True e'_def * show ?thesis by auto
        qed
      qed
    qed
  next
    define e' where "e' = updateEnv x (Storage tp) (Storeloc x) e"
    case f1: False
    with s1 v1 have *: "init ct x e = e'" using e'_def by auto
    then show ?thesis
    proof (cases "fmlookup ct y")
      case None
      then show ?thesis by simp
    next
      case s3: (Some a)
      then show ?thesis
      proof (cases a)
        case (Method _)
        with s3 show ?thesis by simp
      next
        case (Function _)
        with s3 show ?thesis by simp
      next
        case v2: (Var tp')
        then show ?thesis
        proof (cases "y |∈| fmdom (denvalue e)")
          case t1: True
          with e'_def have "y |∈| fmdom (denvalue e')" by simp
          with s1 s3 v1 f1 v2 show ?thesis using e'_def by fastforce
        next
          define f' where "f' = updateEnv y (Storage tp') (Storeloc y) e"
          define e'' where "e'' = updateEnv y (Storage tp') (Storeloc y) e'"
          case f2: False
          with s3 v2 have **: "init ct y e = f'" using f'_def by auto
          show ?thesis
```

```
                proof (cases "y = x")
                  case True
                  with s3 v2 e'_def have "init ct y e' = e''" by simp
                  moreover from s3 v2 True f'_def have "init ct x f' = f''" by simp
                  ultimately show ?thesis using True by simp
                next
                  define f'' where "f'' = updateEnv x (Storage tp) (Storeloc x) f'"
                  case f3: False
                  with f2 have "¬ y |∈| fmdom (denvalue e')" using e'_def by simp
                  with s3 v2 e''_def have "init ct y e' = e''" by auto
                  with * have "(init ct y ∘ init ct x) e = e''" by simp
                  moreover have "init ct x f' = f''"
                  proof -
                    from s1 v1 have "init ct x f' = updateEnvDup x (Storage tp) (Storeloc x) f'" by
simp
                    moreover from f1 f3 have "x |∉| fmdom (denvalue f')" using f'_def by simp
                    ultimately show ?thesis using f''_def by auto
                  qed
                  moreover from f''_def e''_def f'_def e'_def f3 have "Some f'' = Some e''" using
env_reorder_neq by simp
                  ultimately show ?thesis using ** by simp
                qed
              qed
            qed
          qed
        qed
      qed
    qed
  qed
qed

lemma init_address[simp]:
  "address (init ct i e) = address e"
proof (cases "fmlookup ct i")
  case None
  then show ?thesis by simp
next
  case (Some a)
  show ?thesis
  proof (cases a)
    case (Method _)
    with Some show ?thesis by simp
  next
    case (Function _)
    with Some show ?thesis by simp
  next
    case (Var _)
    with Some show ?thesis using updateEnvDup_address updateEnvDup_sender by simp
  qed
qed

lemma init_sender[simp]:
"sender (init ct i e) = sender e"
proof (cases "fmlookup ct i")
  case None
  then show ?thesis by simp
next
  case (Some a)
  show ?thesis
  proof (cases a)
    case (Method _)
    with Some show ?thesis by simp
  next
    case (Function _)
```

```
    with Some show ?thesis by simp
  next
    case (Var _)
    with Some show ?thesis using updateEnvDup_sender by simp
  qed
qed

lemma init_svalue[simp]:
"svalue (init ct i e) = svalue e"
proof (cases "fmlookup ct i")
  case None
  then show ?thesis by simp
next
  case (Some a)
  show ?thesis
  proof (cases a)
    case (Method _)
    with Some show ?thesis by simp
  next
    case (Function _)
    with Some show ?thesis by simp
  next
    case (Var _)
    with Some show ?thesis using updateEnvDup_svalue by simp
  qed
qed

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 (induct xs)
  case empty
  then show ?case by (simp add: ffold_def)
next
  case (insert x xs)
  then have *: "ffold (init ct) e (finsert x xs) =
    init ct x (ffold (init ct) e xs)" using FSet.comp_fun_commute.ffold_finsert[OF init_commte] by simp
  show ?case
  proof (rule allI[OF impI])
    fix e' assume **: "ffold (init ct) e (finsert x xs) = e'"
    with * obtain e'' where ***: "ffold (init ct) e xs = e''" by simp
    with insert have "address e'' = address e ∧ sender e'' = sender e ∧ svalue e'' = svalue e" by
blast
    with * ** *** show "address e' = address e ∧ sender e' = sender e ∧ svalue e' = svalue e" using
init_address init_sender init_svalue by metis
  qed
qed

lemma ffold_init_ad[simp]: "address (ffold (init ct) e xs) = address e"
  using ffold_init_ad_same by simp

lemma ffold_init_sender[simp]: "sender (ffold (init ct) e xs) = sender e"
  using ffold_init_ad_same by simp

lemma ffold_init_dom:
  "fmdom (denvalue (ffold (init ct) e xs)) |⊆| fmdom (denvalue e) |∪| xs"
proof (induct "xs")
  case empty
  then show ?case
  proof
    fix x
    assume "x |∈| fmdom (denvalue (ffold (init ct) e {||}))"
    moreover have "ffold (init ct) e {||} = e" using FSet.comp_fun_commute.ffold_empty[OF init_commte,
of ct e] by simp
    ultimately show "x |∈| fmdom (denvalue e) |∪| {||}" by simp
```

```
  qed
next
  case (insert x xs)
  then have *: "ffold (init ct) e (finsert x xs) =
    init ct x (ffold (init ct) e xs)" using FSet.comp_fun_commute.ffold_finsert[OF init_commte] by simp

  show ?case
  proof
    fix x' assume "x' |∈| fmdom (denvalue (ffold (init ct) e (finsert x xs)))"
    with * have **: "x' |∈| fmdom (denvalue (init ct x (ffold (init ct) e xs)))" by simp
    then consider "x' |∈| fmdom (denvalue (ffold (init ct) e xs))" | "x'=x"
    proof (cases "fmlookup ct x")
      case None
      then show ?thesis using that ** by simp
    next
      case (Some a)
      then show ?thesis
      proof (cases a)
        case (Method _)
        then show ?thesis using Some ** that by simp
      next
        case (Function _)
        then show ?thesis using Some ** that by simp
      next
        case (Var x2)
        show ?thesis
        proof (cases "x=x'")
          case True
          then show ?thesis using that by simp
        next
          case False
          then have "fmlookup (denvalue (updateEnvDup x (Storage x2) (Storeloc x) (ffold (init ct) e
xs))) x' = fmlookup (denvalue (ffold (init ct) e xs)) x'" using updateEnvDup_dup by simp
          moreover from ** Some Var have ***:"x' |∈| fmdom (denvalue (updateEnvDup x (Storage x2)
(Storeloc x) (ffold (init ct) e xs)))" by simp
          ultimately have "x' |∈| fmdom (denvalue (ffold (init ct) e xs))" by (simp add:
fmlookup_dom_iff)
          then show ?thesis using that by simp
        qed
      qed
    qed
    then show "x' |∈| fmdom (denvalue e) |∪| finsert x xs"
    proof cases
      case 1
      then show ?thesis using insert.hyps by auto
    next
      case 2
      then show ?thesis by simp
    qed
  qed
qed

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 (induct "xs")
  case empty
  then show ?case by simp
next
  case (insert x xs)
  then have *: "ffold (init ct) e (finsert x xs) =
    init ct x (ffold (init ct) e xs)" using FSet.comp_fun_commute.ffold_finsert[OF init_commte] by simp
```

```
  from insert.prems consider (a) "i |∈| xs" | (b) "¬ i |∈| xs ∧ i = x" by auto
  then show "fmlookup (denvalue (ffold (init ct) e (finsert x xs))) i = Some (Storage tp, Storeloc i)"
  proof cases
    case a
    with insert.hyps(2) have "fmlookup (denvalue (ffold (init ct) e xs)) i = Some (Storage tp, Storeloc
i)" by simp
    moreover have "fmlookup (denvalue (init ct x (ffold (init ct) e xs))) i = fmlookup (denvalue
(ffold (init ct) e xs)) i"
    proof (cases "fmlookup ct x")
      case None
      then show ?thesis by simp
    next
      case (Some a)
      then show ?thesis
      proof (cases a)
        case (Method _)
        with Some show ?thesis by simp
      next
        case (Function _)
        with Some show ?thesis by simp
      next
        case (Var x2)
        with Some have "init ct x (ffold (init ct) e xs) = updateEnvDup x (Storage x2) (Storeloc x)
(ffold (init ct) e xs)" using init_def[of ct x "(ffold (init ct) e xs)"] by simp
        moreover from insert a have "i≠x" by auto
        then have "fmlookup (denvalue (updateEnvDup x (Storage x2) (Storeloc x) (ffold (init ct) e
xs))) i = fmlookup (denvalue (ffold (init ct) e xs)) i" using updateEnvDup_dup[of x i] by simp
        ultimately show ?thesis by simp
      qed
    qed
    ultimately show ?thesis using * by simp
  next
    case b
    with assms(1) have "fmlookup ct x = Some (Var tp)" by simp
    moreover from b assms(2) have "¬ x |∈| fmdom (denvalue (ffold (init ct) e xs))" using
ffold_init_dom by auto
    ultimately have "init ct x (ffold (init ct) e xs) = updateEnv x (Storage tp) (Storeloc x) (ffold
(init ct) e xs)" by auto
    with b * show ?thesis by simp
  qed
qed

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 -
  from assms(1) have "i |∈| fmdom ct" by (rule Finite_Map.fmdomI)
  then show ?thesis using ffold_init_fmap[OF assms] by simp
qed
```

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›
  using  comp_fun_commute_on.fold_set_fold_remdups ffold.rep_eq
           ffold_init_def init_commte sorted_list_of_fset.rep_eq
           sorted_list_of_fset_simps(1)
  by (metis comp_fun_commute.comp_fun_commute comp_fun_commute_on.intro order_refl)
```

```
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"
  using assms by (cases x, auto)

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"
  using assms by (cases x, auto)

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"
  using assms by (cases x, auto)

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"
  using assms by (cases x, auto)

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"
  using assms by (cases x, auto)

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"
  using assms by (cases x, auto)

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"
  using assms by (cases x, auto)
```

```
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"
  using assms by (cases x, auto)

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"
  unfolding lift_def using assms by (auto split: prod.split_asm result.split option.split_asm
option.split Stackvalue.split Type.split)

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"
  by pat_completeness auto
```

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 (cases "expr e1 e cd st g")
  case (n a g0')
  then show ?thesis
  proof (cases a)
    case (Pair b c)
    then show ?thesis
    proof (cases b)
      case (KValue v1)
      then show ?thesis
      proof (cases c)
        case (Value t1)
        then show ?thesis
        proof (cases "expr e2 e cd st g0'")
          case r2: (n a' g0'')
          then show ?thesis
          proof (cases a')
            case p2: (Pair b c)
            then show ?thesis
            proof (cases b)
              case v2: (KValue v2)
              then show ?thesis
              proof (cases c)
                case t2: (Value t2)
                then show ?thesis
                proof (cases "f t1 t2 v1 v2")
                  case None
                  with assms n Pair KValue Value r2 p2 v2 t2 show ?thesis by simp
                next
                  case (Some a'')
                  then show ?thesis
                  proof (cases a'')
                    case p3: (Pair v t)
                    with assms n Pair KValue Value r2 p2 v2 t2 Some have "g0'≤ g" by simp
                    moreover from assms n Pair KValue Value r2 p2 v2 t2 Some have "g0''≤g0'" by simp
                    moreover from assms n Pair KValue Value r2 p2 v2 t2 Some have "g'=g0''" by (simp
split:prod.split_asm)
                    ultimately show ?thesis by arith
                  qed
                qed
              next
                case (Calldata x2)
                with assms n Pair KValue Value r2 p2 v2 show ?thesis by simp
              next
                case (Memory x3)
                with assms n Pair KValue Value r2 p2 v2 show ?thesis by simp
```

```
            next
              case (Storage x4)
              with assms n Pair KValue Value r2 p2 v2 show ?thesis by simp
            qed
          next
            case (KCDptr x2)
            with assms n Pair KValue Value r2 p2 show ?thesis by simp
          next
            case (KMemptr x3)
            with assms n Pair KValue Value r2 p2 show ?thesis by simp
          next
            case (KStoptr x4)
            with assms n Pair KValue Value r2 p2 show ?thesis by simp
          qed
        qed
      next
        case (e x)
        with assms n Pair KValue Value show ?thesis by simp
      qed
    next
      case (Calldata x2)
      with assms n Pair KValue show ?thesis by simp
    next
      case (Memory x3)
      with assms n Pair KValue show ?thesis by simp
    next
      case (Storage x4)
      with assms n Pair KValue show ?thesis by simp
    qed
  next
    case (KCDptr x2)
    with assms n Pair show ?thesis by simp
  next
    case (KMemptr x3)
    with assms n Pair show ?thesis by simp
  next
    case (KStoptr x4)
    with assms n Pair show ?thesis by simp
  qed
qed
next
  case (e x)
  with assms show ?thesis by simp
qed

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 (induct rule: msel_ssel_expr_load_rexp.pinduct
[where ?P1.0="λc1 t1 l1 xe1 ev1 cd1 st1 g1. (∀l1' g1'. msel c1 t1 l1 xe1 ev1 cd1 st1 g1 = Normal (l1',
g1') ⟶ g1' ≤ g1)"
   and ?P2.0="λt2 l2 xe2 ev2 cd2 st2 g2. (∀v2' g2'. ssel t2 l2 xe2 ev2 cd2 st2 g2 = Normal (v2', g2')
⟶ g2' ≤ g2)"
   and ?P3.0="λe4 ev4 cd4 st4 g4. (∀v4' g4'. expr e4 ev4 cd4 st4 g4 = Normal (v4', g4') ⟶ g4' ≤ g4)"
```

```
   and ?P4.0="λlcp 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)"
   and ?P5.0="λl3 ev3 cd3 st3 g3. (∀ v3' g3'. rexp l3 ev3 cd3 st3 g3 = Normal (v3', g3') ⟶ g3' ≤ g3)"
])
  case 1
  then show ?case using msel.psimps(1) by auto
next
  case 2
  then show ?case using msel.psimps(2) by auto
next
  case 3
  then show ?case using msel.psimps(3) by (auto split: if_split_asm Type.split_asm
Stackvalue.split_asm prod.split_asm StateMonad.result.split_asm)
next
  case (4 mm al t loc x y ys env cd st g)
  show ?case
  proof (rule allI[THEN allI, OF impI])
    fix v1' g1' assume a1: "msel mm (MTArray al t) loc (x # y # ys) env cd st g = Normal (v1', g1')"
    show "g1' ≤ g"
    proof (cases v1')
      case (Pair l1' t1')
      then show ?thesis
      proof (cases "expr x env cd st g")
        case (n a g')
        then show ?thesis
        proof (cases a)
          case p2: (Pair b c)
          then show ?thesis
          proof (cases b)
            case (KValue v)
            then show ?thesis
            proof (cases c)
              case (Value t')
              then show ?thesis
              proof (cases)
                assume l: "less t' (TUInt 256) v (ShowL_int al) = Some (ShowL_bool True, TBool)"
                then show ?thesis
                proof (cases "accessStore (hash loc v) (if mm then memory st else cd)")
                  case None
                  with 4 a1 n p2 KValue Value l show ?thesis using msel.psimps(4) by simp
                next
                  case (Some a)
                  then show ?thesis
                  proof (cases a)
                    case (MValue _)
                    with 4 a1 n p2 KValue Value Some l show ?thesis using msel.psimps(4) by simp
                  next
                    case (MPointer l)
                    with n p2 KValue Value l Some
                    have "msel mm (MTArray al t) loc (x # y # ys) env cd st g = msel mm t l (y # ys)
env cd st g'"
                      using msel.psimps(4) 4(1) by simp
                    moreover from n have "g' ≤ g" using 4(2) by simp
                    moreover from a1 MPointer n Pair p2 KValue Value l Some
                    have "g1' ≤ g'" using msel.psimps(4) 4(3) 4(1) by simp
                    ultimately show ?thesis by simp
                  qed
                qed
              next
                assume "¬ less t' (TUInt 256) v (ShowL_int al) = Some (ShowL_bool True, TBool)"
                with 4 a1 n p2 KValue Value show ?thesis using msel.psimps(4) by simp
              qed
            next
```

```
            case (Calldata _)
            with 4 a1 n p2 KValue show ?thesis using msel.psimps(4) by simp
          next
            case (Memory _)
            with 4 a1 n p2 KValue show ?thesis using msel.psimps(4) by simp
          next
            case (Storage _)
            with 4 a1 n p2 KValue show ?thesis using msel.psimps(4) by simp
          qed
        next
          case (KCDptr _)
          with 4 a1 n p2 show ?thesis using msel.psimps(4) by simp
        next
          case (KMemptr _)
          with 4 a1 n p2 show ?thesis using msel.psimps(4) by simp
        next
          case (KStoptr _)
          with 4 a1 n p2 show ?thesis using msel.psimps(4) by simp
        qed
      qed
    next
      case (e _)
      with 4 a1 show ?thesis using msel.psimps(4) by simp
    qed
  qed
qed
  case 5
  then show ?case using ssel.psimps(1) by auto
next
  case 6
  then show ?case using ssel.psimps(2) by auto
next
  case (7 al t loc x xs env cd st g)
  show ?case
  proof (rule allI[THEN allI, OF impI])
    fix v2' g2' assume a1: "ssel (STArray al t) loc (x # xs) env cd st g = Normal (v2', g2')"
    show "g2' ≤ g"
    proof (cases v2')
      case (Pair l2' t2')
      then show ?thesis
      proof (cases "expr x env cd st g")
        case (n a g'')
        then show ?thesis
        proof (cases a)
          case p2: (Pair b c)
          then show ?thesis
          proof (cases b)
            case (KValue v)
            then show ?thesis
            proof (cases c)
              case (Value t')
              then show ?thesis
              proof (cases)
                assume l: "less t' (TUInt 256) v (ShowL_int al) = Some (ShowL_bool True, TBool)"
                with n p2 KValue Value l
                have "ssel (STArray al t) loc (x # xs) env cd st g = ssel t (hash loc v) xs env cd st
g''"
                using ssel.psimps(3) 7(1) by simp
                moreover from n have "g'' ≤ g" using 7(2) by simp
                moreover from a1 n Pair p2 KValue Value l
                have "g2' ≤ g''" using ssel.psimps(3) 7(3) 7(1) by simp
                ultimately show ?thesis by simp
              next
```

```
            assume "¬ less t' (TUInt 256) v (ShowL_int al) = Some (ShowL_bool True, TBool)"
            with 7 a1 n p2 KValue Value show ?thesis using ssel.psimps(3) by simp
          qed
        next
          case (Calldata _)
          with 7 a1 n p2 KValue show ?thesis using ssel.psimps(3) by simp
        next
          case (Memory _)
          with 7 a1 n p2 KValue show ?thesis using ssel.psimps(3) by simp
        next
          case (Storage _)
          with 7 a1 n p2 KValue show ?thesis using ssel.psimps(3) by simp
        qed
      next
        case (KCDptr _)
        with 7 a1 n p2 show ?thesis using ssel.psimps(3) by simp
      next
        case (KMemptr _)
        with 7 a1 n p2 show ?thesis using ssel.psimps(3) by simp
      next
        case (KStoptr x4)
        with 7 a1 n p2 show ?thesis using ssel.psimps(3) by simp
      qed
    qed
  next
    case (e _)
    with 7 a1 show ?thesis using ssel.psimps(3) by simp
  qed
 qed
qed
next
 case (8 vv t loc x xs env cd st g)
 show ?case
 proof (rule allI[THEN allI, OF impI])
  fix v2' g2' assume a1: "ssel (STMap vv t) loc (x # xs) env cd st g = Normal (v2', g2')"
  show "g2' ≤ g"
  proof (cases v2')
    case (Pair l2' t2')
    then show ?thesis
    proof (cases "expr x env cd st g")
      case (n a g')
      then show ?thesis
      proof (cases a)
        case p2: (Pair b c)
        then show ?thesis
        proof (cases b)
          case (KValue v)
          with 8 n p2 have "ssel (STMap vv t) loc (x # xs) env cd st g = ssel t (hash loc v) xs env
cd st g'" using ssel.psimps(4) by simp
          moreover from n have "g' ≤ g" using 8(2) by simp
          moreover from a1 n Pair p2 KValue
          have "g2' ≤ g'" using ssel.psimps(4) 8(3) 8(1) by simp
          ultimately show ?thesis by simp
        next
          case (KCDptr _)
          with 8 a1 n p2 show ?thesis using ssel.psimps(4) by simp
        next
          case (KMemptr _)
          with 8 a1 n p2 show ?thesis using ssel.psimps(4) by simp
        next
          case (KStoptr _)
          with 8 a1 n p2 show ?thesis using ssel.psimps(4) by simp
        qed
      qed
```

```
      next
        case (e _)
        with 8 a1 show ?thesis using ssel.psimps(4) by simp
      qed
    qed
  qed
next
  case 9
  then show ?case using expr.psimps(1) by (simp split:if_split_asm)
next
  case 10
  then show ?case using expr.psimps(2) by (simp split:if_split_asm)
next
  case 11
  then show ?case using expr.psimps(3) by simp
next
  case (12 ad e cd st g)
  define gc where "gc = costs_e (BALANCE ad) e cd st"
  show ?case
  proof (rule allI[THEN allI, OF impI])
    fix g4 xa
    assume *: "expr (BALANCE ad) e cd st g = Normal (xa, g4)"
    show "g4 ≤ g"
    proof (cases)
      assume "g ≤ gc"
      with 12 gc_def * show ?thesis using expr.psimps(4) by simp
    next
      assume gcost: "¬ g ≤ gc"
      then show ?thesis
      proof (cases "expr ad e cd st (g - gc)")
        case (n a s)
        show ?thesis
        proof (cases a)
          case (Pair b c)
          then show ?thesis
          proof (cases b)
            case (KValue x1)
            then show ?thesis
            proof (cases c)
              case (Value x1)
              then show ?thesis
              proof (cases x1)
                case (TSInt _)
                with 12 gc_def * gcost n Pair KValue Value show ?thesis using expr.psimps(4)[of ad e
cd st] by simp
              next
                case (TUInt _)
                with 12 gc_def * gcost n Pair KValue Value show ?thesis using expr.psimps(4)[of ad e
cd st] by simp
              next
                case TBool
                with 12 gc_def * gcost n Pair KValue Value show ?thesis using expr.psimps(4)[of ad e
cd st] by simp
              next
                case TAddr
                with 12(2)[where ?s'a="g-costs_e (BALANCE ad) e cd st"] gc_def * gcost n Pair KValue
Value show "g4 ≤ g" using expr.psimps(4)[OF 12(1)] by simp
              qed
            next
              case (Calldata _)
              with 12 gc_def * gcost n Pair KValue show ?thesis using expr.psimps(4)[of ad e cd st]
by simp
            next
              case (Memory _)
```

```
            with 12 gc_def * gcost n Pair KValue show ?thesis using expr.psimps(4)[of ad e cd st]
by simp
          next
            case (Storage _)
            with 12 gc_def * gcost n Pair KValue show ?thesis using expr.psimps(4)[of ad e cd st]
by simp
          qed
        next
          case (KCDptr _)
          with 12 gc_def * gcost n Pair show ?thesis using expr.psimps(4)[of ad e cd st] by simp
        next
          case (KMemptr _)
          with 12 gc_def * gcost n Pair show ?thesis using expr.psimps(4)[of ad e cd st] by simp
        next
          case (KStoptr _)
          with 12 gc_def * gcost n Pair show ?thesis using expr.psimps(4)[of ad e cd st] by simp
        qed
      qed
    next
      case (e _)
      with 12 gc_def * gcost show ?thesis using expr.psimps(4)[of ad e cd st] by simp
    qed
  qed
next
  case 13
  then show ?case using expr.psimps(5) by simp
next
  case 14
  then show ?case using expr.psimps(6) by simp
next
  case 15
  then show ?case using expr.psimps(7) by simp
next
  case 16
  then show ?case using expr.psimps(8) by simp
next
  case 17
  then show ?case using expr.psimps(9) by simp
next
  case (18 x e cd st g)
  define gc where "gc = costs_e (NOT x) e cd st"
  show ?case
  proof (rule allI[THEN allI, OF impI])
    fix v4 g4' assume a1: "expr (NOT x) e cd st g = Normal (v4, g4')"
    show "g4' ≤ g"
    proof (cases v4)
      case (Pair l4 t4)
      show ?thesis
      proof (cases)
        assume "g ≤ gc"
        with gc_def a1 show ?thesis using expr.psimps(10)[OF 18(1)] by simp
      next
        assume gcost: "¬ g ≤ gc"
        then show ?thesis
        proof (cases "expr x e cd st (g - gc)")
          case (n a g')
          then show ?thesis
          proof (cases a)
            case p2: (Pair b c)
            then show ?thesis
            proof (cases b)
              case (KValue v)
              then show ?thesis
```

```
            proof (cases c)
              case (Value t)
              then show ?thesis
              proof (cases t)
                case (TSInt x1)
                with a1 gc_def gcost n p2 KValue Value show ?thesis using expr.psimps(10)[OF 18(1)]
by simp
              next
                case (TUInt x2)
                with a1 gc_def gcost n p2 KValue Value show ?thesis using expr.psimps(10)[OF 18(1)]
by simp
              next
                case TBool
                then show ?thesis
                proof (cases)
                  assume v_def: "v = ShowL_bool True"
                  with 18(1) gc_def gcost n p2 KValue Value TBool have "expr (NOT x) e cd st g = expr
FALSE e cd st g' " using expr.psimps(10)[OF 18(1)] by simp
                  moreover from 18(2) gc_def gcost n p2 have "g' ≤ g-gc" by simp
                  moreover from 18(3)[OF _ _ n] a1 gc_def gcost have "g4' ≤ g'" using
expr.psimps(10)[OF 18(1)] n Pair p2 KValue Value TBool v_def by simp
                  ultimately show ?thesis by arith
                 next
                  assume v_def: "¬ v = ShowL_bool True"
                  then show ?thesis
                  proof (cases)
                    assume v_def2: "v = ShowL_bool False"
                    with 18(1) gc_def gcost n p2 KValue Value v_def TBool have "expr (NOT x) e cd st
g = expr TRUE e cd st g'" using expr.psimps(10) by simp
                    moreover from 18(2)[where ?s'a="g-costs_e (NOT x) e cd st"] gc_def gcost n p2
have "g' ≤ g" by simp
                    moreover from 18(4)[OF _ _ n] a1 gc_def gcost have "g4' ≤ g'" using
expr.psimps(10)[OF 18(1)] n Pair p2 KValue Value TBool v_def v_def2 by simp
                    ultimately show ?thesis by arith
                  next
                    assume "¬ v = ShowL_bool False"
                    with 18(1) a1 gc_def gcost n p2 KValue Value v_def TBool show ?thesis using
expr.psimps(10) by simp
                    qed
                  qed
                next
                  case TAddr
                  with 18(1) a1 gc_def gcost n p2 KValue Value show ?thesis using expr.psimps(10) by
simp
                qed
              next
                case (Calldata _)
                with 18(1) a1 gc_def gcost n p2 KValue show ?thesis using expr.psimps(10) by simp
              next
                case (Memory _)
                with 18(1) a1 gc_def gcost n p2 KValue show ?thesis using expr.psimps(10) by simp
              next
                case (Storage _)
                with 18(1) a1 gc_def gcost n p2 KValue show ?thesis using expr.psimps(10) by simp
              qed
            next
              case (KCDptr _)
              with 18(1) a1 gc_def gcost n p2 show ?thesis using expr.psimps(10) by simp
            next
              case (KMemptr _)
              with 18(1) a1 gc_def gcost n p2 show ?thesis using expr.psimps(10) by simp
            next
              case (KStoptr _)
              with 18(1) a1 gc_def gcost n p2 show ?thesis using expr.psimps(10) by simp
```

```
          qed
        qed
      next
        case (e _)
        with 18(1) a1 gc_def gcost show ?thesis using expr.psimps(10) by simp
      qed
    qed
  qed
qed
next
  case (19 e1 e2 e cd st g)
  define gc where "gc = costs_e (PLUS e1 e2) e cd st"
  show ?case
  proof (rule allI[THEN allI, OF impI])
    fix g4 xa assume e_def: "expr (PLUS e1 e2) e cd st g = Normal (xa, g4)"
    then show "g4 ≤ g"
    proof (cases)
      assume "g ≤ gc"
      with 19(1) e_def show ?thesis using expr.psimps(11) gc_def by simp
    next
      assume "¬ g ≤ gc"
      then have *: "assert Gas ((<) (costs_e (PLUS e1 e2) e cd st)) g = Normal ((), g)" using gc_def by
simp
      with 19(1) e_def gc_def have lift:"lift expr add e1 e2 e cd st (g - gc) = Normal (xa, g4)" using
expr.psimps(11)[OF 19(1)] by simp
      moreover have **: "modify (λg. g - costs_e (PLUS e1 e2) e cd st) g = Normal ((), g - costs_e (PLUS
e1 e2) e cd st)" by simp
      with 19(2)[OF * **] have "⋀g4' v4 t4. expr e1 e cd st (g-gc) = Normal ((v4, t4), g4') ⟹ g4' ≤
g - gc" unfolding gc_def by simp
      moreover obtain v g' where ***: "expr e1 e cd st (g - costs_e (PLUS e1 e2) e cd st) = Normal (v,
g')" using expr.psimps(11)[OF 19(1)] e_def by (simp split:if_split_asm result.split_asm prod.split_asm)
      with 19(3)[OF * ** ***] have "⋀v t g' v' t' g''. expr e1 e cd st (g-gc) = Normal ((KValue v,
Value t), g') ⟹ expr e2 e cd st g' = Normal ((v', t'), g'') ⟹ g'' ≤ g'" unfolding gc_def by simp
      ultimately show "g4 ≤ g" using lift_gas[OF lift] `¬ g ≤ gc` by (simp split:result.split_asm
Stackvalue.split_asm Type.split_asm prod.split_asm)
    qed
  qed
next
  case (20 e1 e2 e cd st g)
  define gc where "gc = costs_e (MINUS e1 e2) e cd st"
  show ?case
  proof (rule allI[THEN allI, OF impI])
    fix g4 xa assume e_def: "expr (MINUS e1 e2) e cd st g = Normal (xa, g4)"
    then show "g4 ≤ g"
    proof (cases)
      assume "g ≤ gc"
      with 20(1) e_def show ?thesis using expr.psimps(12) gc_def by simp
    next
      assume "¬ g ≤ gc"
      then have *: "assert Gas ((<) (costs_e (MINUS e1 e2) e cd st)) g = Normal ((), g)" using gc_def
by simp
      with 20(1) e_def gc_def have lift:"lift expr sub e1 e2 e cd st (g - gc) = Normal (xa, g4)" using
expr.psimps(12)[OF 20(1)] by simp
      moreover have **: "modify (λg. g - costs_e (MINUS e1 e2) e cd st) g = Normal ((), g - costs_e
(MINUS e1 e2) e cd st)" by simp
      with 20(2)[OF * **] have "⋀g4' v4 t4. expr e1 e cd st (g-gc) = Normal ((v4, t4), g4') ⟹ g4' ≤
g - gc" unfolding gc_def by simp
      moreover obtain v g' where ***: "expr e1 e cd st (g - costs_e (MINUS e1 e2) e cd st) =
Normal (v, g')" using expr.psimps(12)[OF 20(1)] e_def by (simp split:if_split_asm result.split_asm
prod.split_asm)
      with 20(3)[OF * ** ***] have "⋀v t g' v' t' g''. expr e1 e cd st (g-gc) = Normal ((KValue v,
Value t), g') ⟹ expr e2 e cd st g' = Normal ((v', t'), g'') ⟹ g'' ≤ g'" unfolding gc_def by simp
      ultimately show "g4 ≤ g" using lift_gas[OF lift] `¬ g ≤ gc` by (simp split:result.split_asm
Stackvalue.split_asm Type.split_asm prod.split_asm)
```

```
    qed
  qed
next
  case (21 e1 e2 e cd st g)
  define gc where "gc = costs_e (LESS e1 e2) e cd st"
  show ?case
  proof (rule allI[THEN allI, OF impI])
    fix g4 xa assume e_def: "expr (LESS e1 e2) e cd st g = Normal (xa, g4)"
    then show "g4 ≤ g"
    proof (cases)
      assume "g ≤ gc"
      with 21(1) e_def show ?thesis using expr.psimps(13) gc_def by simp
    next
      assume "¬ g ≤ gc"
      then have *: "assert Gas ((<) (costs_e (LESS e1 e2) e cd st)) g = Normal ((), g)" using gc_def by
simp
      with 21(1) e_def gc_def have lift:"lift expr less e1 e2 e cd st (g - gc) = Normal (xa, g4)" us-
ing expr.psimps(13)[OF 21(1)] by simp
      moreover have **: "modify (λg. g - costs_e (LESS e1 e2) e cd st) g = Normal ((), g - costs_e (LESS
e1 e2) e cd st)" by simp
      with 21(2)[OF * **] have "⋀g4' v4 t4. expr e1 e cd st (g-gc) = Normal ((v4, t4), g4') ⟹ g4' ≤
g - gc" unfolding gc_def by simp
      moreover obtain v g' where ***: "expr e1 e cd st (g - costs_e (LESS e1 e2) e cd st) = Normal (v,
g')" using expr.psimps(13)[OF 21(1)] e_def by (simp split:if_split_asm result.split_asm prod.split_asm)
      with 21(3)[OF * ** ***] have "⋀v t g' v' t' g''. expr e1 e cd st (g-gc) = Normal ((KValue v,
Value t), g') ⟹ expr e2 e cd st g' = Normal ((v', t'), g'') ⟹ g'' ≤ g'" unfolding gc_def by simp
      ultimately show "g4 ≤ g" using lift_gas[OF lift] `¬ g ≤ gc` by (simp split:result.split_asm
Stackvalue.split_asm Type.split_asm prod.split_asm)
    qed
  qed
next
  case (22 e1 e2 e cd st g)
  define gc where "gc = costs_e (EQUAL e1 e2) e cd st"
  show ?case
  proof (rule allI[THEN allI, OF impI])
    fix g4 xa assume e_def: "expr (EQUAL e1 e2) e cd st g = Normal (xa, g4)"
    then show "g4 ≤ g"
    proof (cases)
      assume "g ≤ gc"
      with 22(1) e_def show ?thesis using expr.psimps(14) gc_def by simp
    next
      assume "¬ g ≤ gc"
      then have *: "assert Gas ((<) (costs_e (EQUAL e1 e2) e cd st)) g = Normal ((), g)" using gc_def
by simp
      with 22(1) e_def gc_def have lift:"lift expr equal e1 e2 e cd st (g - gc) = Normal (xa, g4)"
using expr.psimps(14)[OF 22(1)] by simp
      moreover have **: "modify (λg. g - costs_e (EQUAL e1 e2) e cd st) g = Normal ((), g - costs_e
(EQUAL e1 e2) e cd st)" by simp
      with 22(2)[OF * **] have "⋀g4' v4 t4. expr e1 e cd st (g-gc) = Normal ((v4, t4), g4') ⟹ g4' ≤
g - gc" unfolding gc_def by simp
      moreover obtain v g' where ***: "expr e1 e cd st (g - costs_e (EQUAL e1 e2) e cd st) =
Normal (v, g')" using expr.psimps(14)[OF 22(1)] e_def by (simp split:if_split_asm result.split_asm
prod.split_asm)
      with 22(3)[OF * ** ***] have "⋀v t g' v' t' g''. expr e1 e cd st (g-gc) = Normal ((KValue v,
Value t), g') ⟹ expr e2 e cd st g' = Normal ((v', t'), g'') ⟹ g'' ≤ g'" unfolding gc_def by simp
      ultimately show "g4 ≤ g" using lift_gas[OF lift] `¬ g ≤ gc` by (simp split:result.split_asm
Stackvalue.split_asm Type.split_asm prod.split_asm)
    qed
  qed
next
  case (23 e1 e2 e cd st g)
  define gc where "gc = costs_e (AND e1 e2) e cd st"
  show ?case
  proof (rule allI[THEN allI, OF impI])
```

```
    fix g4 xa assume e_def: "expr (AND e1 e2) e cd st g = Normal (xa, g4)"
    then show "g4 ≤ g"
    proof (cases)
      assume "g ≤ gc"
      with 23(1) e_def show ?thesis using expr.psimps(15) gc_def by simp
    next
      assume "¬ g ≤ gc"
      then have *: "assert Gas ((<) (costs_e (AND e1 e2) e cd st)) g = Normal ((), g)" using gc_def by
simp
      with 23(1) e_def gc_def have lift:"lift expr vtand e1 e2 e cd st (g - gc) = Normal (xa, g4)"
using expr.psimps(15)[OF 23(1)] by simp
      moreover have **: "modify (λg. g - costs_e (AND e1 e2) e cd st) g = Normal ((), g - costs_e (AND
e1 e2) e cd st)" by simp
      with 23(2)[OF * **] have "⋀g4' v4 t4. expr e1 e cd st (g-gc) = Normal ((v4, t4), g4') ⟹ g4' ≤
g - gc" unfolding gc_def by simp
      moreover obtain v g' where ***: "expr e1 e cd st (g - costs_e (AND e1 e2) e cd st) = Normal (v,
g')" using expr.psimps(15)[OF 23(1)] e_def by (simp split:if_split_asm result.split_asm prod.split_asm)
      with 23(3)[OF * ** ***] have "⋀v t g' v' t' g''. expr e1 e cd st (g-gc) = Normal ((KValue v,
Value t), g') ⟹ expr e2 e cd st g' = Normal ((v', t'), g'') ⟹ g'' ≤ g'" unfolding gc_def by simp
      ultimately show "g4 ≤ g" using lift_gas[OF lift] ‘¬ g ≤ gc‘ by (simp split:result.split_asm
Stackvalue.split_asm Type.split_asm prod.split_asm)
    qed
  qed
next
  case (24 e1 e2 e cd st g)
  define gc where "gc = costs_e (OR e1 e2) e cd st"
  show ?case
  proof (rule allI[THEN allI, OF impI])
    fix g4 xa assume e_def: "expr (OR e1 e2) e cd st g = Normal (xa, g4)"
    then show "g4 ≤ g"
    proof (cases)
      assume "g ≤ gc"
      with 24(1) e_def show ?thesis using expr.psimps(16) gc_def by simp
    next
      assume "¬ g ≤ gc"
      then have *: "assert Gas ((<) (costs_e (OR e1 e2) e cd st)) g = Normal ((), g)" using gc_def by
simp
      with 24(1) e_def gc_def have lift:"lift expr vtor e1 e2 e cd st (g - gc) = Normal (xa, g4)" us-
ing expr.psimps(16)[OF 24(1)] by simp
      moreover have **: "modify (λg. g - costs_e (OR e1 e2) e cd st) g = Normal ((), g - costs_e (OR e1
e2) e cd st)" by simp
      with 24(2)[OF * **] have "⋀g4' v4 t4. expr e1 e cd st (g-gc) = Normal ((v4, t4), g4') ⟹ g4' ≤
g - gc" unfolding gc_def by simp
      moreover obtain v g' where ***: "expr e1 e cd st (g - costs_e (OR e1 e2) e cd st) = Normal (v,
g')" using expr.psimps(16)[OF 24(1)] e_def by (simp split:if_split_asm result.split_asm prod.split_asm)
      with 24(3)[OF * ** ***] have "⋀v t g' v' t' g''. expr e1 e cd st (g-gc) = Normal ((KValue v,
Value t), g') ⟹ expr e2 e cd st g' = Normal ((v', t'), g'') ⟹ g'' ≤ g'" unfolding gc_def by simp
      ultimately show "g4 ≤ g" using lift_gas[OF lift] ‘¬ g ≤ gc‘ by (simp split:result.split_asm
Stackvalue.split_asm Type.split_asm prod.split_asm)
    qed
  qed
next
  case (25 i e cd st g)
  define gc where "gc = costs_e (LVAL i) e cd st"
  show ?case
  proof (rule allI[THEN allI, OF impI])
    fix g4 xa xaa assume e_def: "expr (LVAL i) e cd st g = Normal (xa, g4)"
    then show "g4 ≤ g"
    proof (cases)
      assume "g ≤ gc"
      with 25(1) e_def show ?thesis using expr.psimps(17) gc_def by simp
    next
      assume "¬ g ≤ gc"
      then have *: "assert Gas ((<) (costs_e (LVAL i) e cd st)) g = Normal ((), g)" using gc_def by
```

```
simp
      then have "expr (LVAL i) e cd st g = rexp i e cd st (g - gc)" using expr.psimps(17)[OF 25(1)]
gc_def by simp
      then have "rexp i e cd st (g - gc) = Normal (xa, g4)" using e_def by simp
      moreover have **: "modify (λg. g - costs_e (LVAL i) e cd st) g = Normal ((), g - costs_e (LVAL i)
e cd st)" by simp
      ultimately show ?thesis using 25(2)[OF * **] unfolding gc_def by (simp split:result.split_asm
Stackvalue.split_asm Type.split_asm prod.split_asm)
    qed
  qed
next
  case (26 i xe e cd st g)
  define gc where "gc = costs_e (CALL i xe) e cd st"
  show ?case
  proof (rule allI[THEN allI, OF impI])
    fix g4' v4 assume a1: "expr (CALL i xe) e cd st g = Normal (v4, g4')"
    then have *: "assert Gas ((<) (costs_e (CALL i xe) e cd st)) g = Normal ((), g)" using gc_def using
expr.psimps(18)[OF 26(1)] by (simp split:if_split_asm)
    have **: "modify (λg. g - costs_e (CALL i xe) e cd st) g = Normal ((), g - gc)" unfolding gc_def by
simp
    show "g4' ≤ g"
    proof (cases)
      assume "g ≤ gc"
      with 26(1) gc_def a1 show ?thesis using expr.psimps(18) by simp
    next
      assume gcost: "¬ g ≤ gc"
      then show ?thesis
      proof (cases v4)
        case (Pair l4 t4)
        then show ?thesis
        proof (cases "ep $$ contract e")
          case None
          with 26(1) a1 gc_def gcost show ?thesis using expr.psimps(18) by simp
        next
          case (Some a)
          then show ?thesis
          proof (cases a)
            case p2: (fields ct x0 x1)
            then have 1: "option Err (λ_. ep $$ contract e) (g - gc) = Normal ((ct, x0, x1), (g - gc))"
using Some by simp
            then show ?thesis
            proof (cases "fmlookup ct i")
              case None
              with 26(1) a1 gc_def gcost Some p2 show ?thesis using expr.psimps(18) by simp
            next
              case s1: (Some a)
              then show ?thesis
              proof (cases a)
                case (Function x1)
                then show ?thesis
                proof (cases x1)
                  case p1: (fields fp ext x)
                  then show ?thesis
                  proof (cases ext)
                    case True
                    with 26(1) a1 gc_def gcost Some p2 s1 Function p1 show ?thesis using
expr.psimps(18)[of i xe e cd st] by (auto split:unit.split_asm)
                  next
                    case False
                    then have 2: "(case ct $$ i of None ⇒ throw Err | Some (Function (fp, True, xa))
⇒ throw Err | Some (Function (fp, False, xa)) ⇒ return (fp, xa) | Some _ ⇒ throw Err) (g - gc) =
Normal ((fp, x), (g - gc))" using s1 Function p1 by simp
                    define e' where "e' = ffold (init ct) (emptyEnv (address e) (contract e) (sender e)
(svalue e)) (fmdom ct)"
```

```
                then show ?thesis
                proof (cases "load False fp xe e' emptyStore emptyStore (memory st) e cd st (g -
gc)")
                  case s4: (n a g')
                  then show ?thesis
                  proof (cases a)
                    case f2: (fields e_l cd_l k_l m_l)
                    then show ?thesis
                    proof (cases "expr x e_l cd_l (st(|stack:=k_l, memory:=m_l|)) g'")
                      case n2: (n a g'')
                      then show ?thesis
                      proof (cases a)
                        case p3: (Pair sv tp)
                        with 26(1) a1 gc_def gcost Some p2 s1 Function p1 e'_def s4 f2 n2 False
                        have "expr (CALL i xe) e cd st g = Normal ((sv, tp), g'')"
                          using expr.psimps(18)[OF 26(1)] by simp
                        with a1 have "g4' ≤ g''" by simp
                        also from 26(3)[OF * ** 1 _ 2 _ _ s4] e'_def f2 n2 p3 gcost gc_def
                          have "... ≤ g'" by auto
                        also from 26(2)[OF * ** 1 _ 2 _] False e'_def f2 s4
                          have "... ≤ g - gc" unfolding ffold_init_def gc_def by blast
                        finally show ?thesis by simp
                      qed
                    next
                      case (e _)
                      with 26(1) a1 gc_def gcost Some p2 s1 Function p1 e'_def s4 f2 False show
?thesis using expr.psimps(18)[of i xe e cd st] by (auto simp add:Let_def split:unit.split_asm)
                    qed
                  qed
                next
                  case (e _)
                  with 26(1) a1 gc_def gcost Some p2 s1 Function p1 e'_def False show ?thesis
using expr.psimps(18)[of i xe e cd st] by (auto split:unit.split_asm)
                qed
              qed
            qed
          next
            case (Method _)
            with 26(1) a1 gc_def gcost Some p2 s1 show ?thesis using expr.psimps(18) by simp
          next
            case (Var _)
            with 26(1) a1 gc_def gcost Some p2 s1 show ?thesis using expr.psimps(18) by simp
          qed
        qed
      qed
    qed
  qed
next
  case (27 ad i xe e cd st g)
  define gc where "gc = costs_e (ECALL ad i xe) e cd st"
  show ?case
  proof (rule allI[THEN allI, OF impI])
    fix g4' v4 assume a1: "expr (ECALL ad i xe) e cd st g = Normal (v4, g4')"
    show "g4' ≤ g"
    proof (cases v4)
      case (Pair l4 t4)
      then show ?thesis
      proof (cases)
        assume "g ≤ gc"
        with gc_def a1 show ?thesis using expr.psimps(19)[OF 27(1)] by simp
      next
        assume gcost: "¬ g ≤ gc"
```

```
        then have *: "assert Gas ((<) (costs_e (ECALL ad i xe) e cd st)) g = Normal ((), g)" using
gc_def using expr.psimps(19)[OF 27(1)] by (simp split:if_split_asm)
        have **: "modify (λg. g - costs_e (ECALL ad i xe) e cd st) g = Normal ((), g - gc)" unfolding
gc_def by simp
        then show ?thesis
        proof (cases "expr ad e cd st (g-gc)")
          case (n a0 g')
          then show ?thesis
          proof (cases a0)
            case p0: (Pair a b)
            then show ?thesis
            proof (cases a)
              case (KValue adv)
              then show ?thesis
              proof (cases b)
                case (Value x1)
                then show ?thesis
                proof (cases x1)
                  case (TSInt x1)
                  with a1 gc_def gcost n p0 KValue Value show ?thesis using expr.psimps(19)[OF 27(1)]
by simp
                next
                  case (TUInt x2)
                  with a1 gc_def gcost n p0 KValue Value show ?thesis using expr.psimps(19)[OF 27(1)]
by simp
                next
                  case TBool
                  with a1 gc_def gcost n p0 KValue Value show ?thesis using expr.psimps(19)[OF 27(1)]
by simp
                next
                  case TAddr
                  then have 1: "(case a0 of (KValue adv, Value TAddr) ⇒ return adv | (KValue adv,
Value _) ⇒ throw Err | (KValue adv, _) ⇒ throw Err | (_, b) ⇒ throw Err) g' = Normal (adv, g')" us-
ing p0 KValue Value by simp
                  then show ?thesis
                  proof (cases "adv = address e")
                    case True
                    with a1 gc_def gcost n p0 KValue Value TAddr show ?thesis using expr.psimps(19)[OF
27(1)] by simp
                  next
                    case False
                    then have 2: "assert Err (λ_. adv ≠ address e) g' = Normal ((), g')" by simp
                    then show ?thesis
                    proof (cases "type (accounts st adv)")
                      case None
                      with a1 gc_def gcost n p0 KValue Value TAddr False show ?thesis using
expr.psimps(19)[OF 27(1)] by simp
                    next
                      case (Some x2)
                      then show ?thesis
                      proof (cases x2)
                        case EOA
                        with a1 gc_def gcost n p0 KValue Value TAddr False Some show ?thesis using
expr.psimps(19)[OF 27(1)] by simp
                      next
                        case c: (Contract c)
                        then have 3: "(case type (accounts st adv) of Some (Contract c) ⇒ return c | _
⇒ throw Err) g' = Normal (c, g')" using Some by simp
                        then show ?thesis
                        proof (cases "ep $$ c")
                          case None
                          with a1 gc_def gcost n p0 KValue Value TAddr False Some c show ?thesis using
expr.psimps(19)[OF 27(1)] by simp
                        next
```

```
                        case s0: (Some a)
                        then show ?thesis
                        proof (cases a)
                          case p1: (fields ct xa xb)
                          then have 4: "option Err (λ_. ep $$ c) g' = Normal ((ct, xa, xb), g')"
using Some s0 by simp
                          then show ?thesis
                          proof (cases "ct $$ i")
                            case None
                            with a1 gc_def gcost n p0 KValue Value TAddr Some p1 False c s0 show
?thesis using expr.psimps(19)[OF 27(1)] by simp
                          next
                            case s1: (Some a)
                            then show ?thesis
                            proof (cases a)
                              case (Function x1)
                              then show ?thesis
                              proof (cases x1)
                                case p2: (fields fp ext x)
                                then show ?thesis
                                proof (cases ext)
                                  case True
                                  then have 5: "(case ct $$ i of None ⇒ throw Err | Some (Function
(fp, True, xa)) ⇒ return (fp, xa) | Some (Function (fp, False, xa)) ⇒ throw Err | Some _ ⇒ throw Err)
g' = Normal ((fp, x), g')" using s1 Function p2 by simp
                                  define e' where "e' = ffold (init ct) (emptyEnv adv c (address e)
(ShowL_nat 0)) (fmdom ct)"
                                  then show ?thesis
                                  proof (cases "load True fp xe e' emptyStore emptyStore emptyStore e
cd st g'")
                                    case n1: (n a g'')
                                    then show ?thesis
                                    proof (cases a)
                                      case f2: (fields e_l cd_l k_l m_l)
                                      show ?thesis
                                      proof (cases "expr x e_l cd_l (st(|stack:=k_l, memory:=m_l|)) g''")
                                        case n2: (n a g''')
                                        then show ?thesis
                                        proof (cases a)
                                          case p3: (Pair sv tp)
                                          with a1 gc_def gcost n p2 KValue Value TAddr Some p1 s1
Function p0 e'_def n1 f2 n2 True False s0 c
                                          have "expr (ECALL ad i xe) e cd st g = Normal ((sv, tp),
g''')"
                                              using expr.psimps(19)[OF 27(1)] by auto
                                          with a1 have "g4' ≤ g'''" by auto
                                          also from 27(4)[OF * ** n 1 2 3 4 _ 5 _ _ n1] p3 f2 e'_def
n2
                                            have "... ≤ g''" by simp
                                          also from 27(3)[OF * ** n 1 2 3 4 _ 5, of "(xa, xb)" fp x
e'] e'_def n1 f2
                                            have "... ≤ g'" by auto
                                          also from 27(2)[OF * **] n
                                            have "... ≤ g" by simp
                                          finally show ?thesis by simp
                                        qed
                                      next
                                        case (e _)
                                        with a1 gc_def gcost n p0 KValue Value TAddr False Some p1 s1
Function p2 e'_def n1 f2 True s0 c show ?thesis using expr.psimps(19)[OF 27(1)] by simp
                                      qed
                                    qed
                                  next
                                    case (e _)
```

```
                                    with a1 gc_def gcost n p0 KValue Value TAddr False Some p1 s1
Function p2 e'_def True s0 c show ?thesis using expr.psimps(19)[OF 27(1)] by simp
                                  qed
                                next
                                  case f: False
                                  with a1 gc_def gcost n p0 KValue Value TAddr Some p1 s1 Function p2
False s0 c show ?thesis using expr.psimps(19)[OF 27(1)] by simp
                                qed
                              qed
                            next
                              case (Method _)
                              with a1 gc_def gcost n p0 KValue Value TAddr Some p1 s1 False s0 c
show ?thesis using expr.psimps(19)[OF 27(1)] by simp
                            next
                              case (Var _)
                              with a1 gc_def gcost n p0 KValue Value TAddr Some p1 s1 False s0 c
show ?thesis using expr.psimps(19)[OF 27(1)] by simp
                            qed
                          qed
                        qed
                      qed
                    qed
                  qed
                qed
              next
                case (Calldata _)
                with a1 gc_def gcost n p0 KValue show ?thesis using expr.psimps(19)[OF 27(1)] by simp
              next
                case (Memory _)
                with a1 gc_def gcost n p0 KValue show ?thesis using expr.psimps(19)[OF 27(1)] by simp
              next
                case (Storage _)
                with a1 gc_def gcost n p0 KValue show ?thesis using expr.psimps(19)[OF 27(1)] by simp
              qed
            next
              case (KCDptr _)
              with a1 gc_def gcost n p0 show ?thesis using expr.psimps(19)[OF 27(1)] by simp
            next
              case (KMemptr _)
              with a1 gc_def gcost n p0 show ?thesis using expr.psimps(19)[OF 27(1)] by simp
            next
              case (KStoptr _)
              with a1 gc_def gcost n p0 show ?thesis using expr.psimps(19)[OF 27(1)] by simp
            qed
          qed
        next
          case (e _)
          with a1 gc_def gcost show ?thesis using expr.psimps(19)[OF 27(1)] by simp
        qed
      qed
    qed
  qed
next
  case (28 cp i_p t_p pl e el e_v' cd' sck' mem' e_v cd st g)
  then show ?case
  proof (cases "expr e e_v cd st g")
    case (n a g'')
    then show ?thesis
    proof (cases a)
      case (Pair v t)
      then show ?thesis
      proof (cases "decl i_p t_p (Some (v,t)) cp cd (memory st) (storage st) (cd', mem',  sck', e_v')")
        case None
```

```
        with 28(1) n Pair show ?thesis using load.psimps(1) by simp
      next
        case (Some a)
        show ?thesis
        proof (cases a)
          case (fields cd'' m'' k'' e_v'')
          then have 1: "(case decl i_p t_p (Some (v, t)) cp cd (memory st) (storage st) (cd',
mem', sck', e_v') of None ⇒ throw Err | Some (c, m, k, e) ⇒ return (c, m, k, e)) g'' = Normal
((cd'',m'',k'',e_v''), g'')" using Some by simp
          show ?thesis
          proof ((rule allI)+,(rule impI))
            fix ev cda k m g' assume load_def: "load cp ((i_p, t_p) # pl) (e # el) e_v' cd' sck' mem' e_v
cd st g = Normal ((ev, cda, k, m), g')"
            with n Pair Some fields have "load cp ((i_p, t_p) # pl) (e # el) e_v' cd' sck' mem' e_v cd st
g = load cp pl el e_v'' cd'' k'' m'' e_v cd st g''" using load.psimps(1)[OF 28(1)] by simp
            with load_def have "load cp pl el e_v'' cd'' k'' m'' e_v cd st g'' = Normal ((ev, cda, k,
m), g')" by simp
            with Pair fields have "g' ≤ g'' ∧ address ev = address e_v'' ∧ sender ev = sender e_v'' ∧
svalue ev = svalue e_v''" using 28(3)[OF n Pair 1, of cd'' _ m''] by simp
            moreover from n have "g'' ≤ g" using 28(2) by simp
            moreover from Some fields have "address e_v'' =  address e_v' ∧ sender e_v'' = sender e_v' ∧
svalue e_v'' = svalue e_v'" using decl_env by auto
            ultimately show "g' ≤ g ∧ address ev = address e_v' ∧ sender ev = sender e_v' ∧ svalue ev =
svalue e_v'" by auto
          qed
        qed
      qed
    qed
  next
    case (e _)
    with 28(1) show ?thesis using load.psimps(1) by simp
  qed
next
  case 29
  then show ?case using load.psimps(2) by auto
next
  case 30
  then show ?case using load.psimps(3) by auto
next
  case 31
  then show ?case using load.psimps(4) by auto
next
  case (32 i e cd st g)
  show ?case
  proof (rule allI[THEN allI, OF impI])
    fix xa xaa assume "rexp (L.Id i) e cd st g = Normal (xa, xaa)"
    then show "xaa ≤ g" using 32(1) rexp.psimps(1) by (simp split: option.split_asm
Denvalue.split_asm Stackvalue.split_asm prod.split_asm Type.split_asm STypes.split_asm)
  qed
next
  case (33 i r e cd st g)
  show ?case
  proof (rule allI[THEN allI,OF impI])
    fix xa xaa assume rexp_def: "rexp (Ref i r) e cd st g = Normal (xa, xaa)"
    show "xaa ≤ g"
    proof (cases "fmlookup (denvalue e) i")
      case None
      with 33(1) show ?thesis using rexp.psimps rexp_def by simp
    next
      case (Some a)
      then show ?thesis
      proof (cases a)
        case (Pair tp b)
        then show ?thesis
```

```
        proof (cases b)
          case (Stackloc l)
          then show ?thesis
          proof (cases "accessStore l (stack st)")
            case None
            with 33(1) Some Pair Stackloc show ?thesis using rexp.psimps(2) rexp_def by simp
          next
            case s1: (Some a)
            then show ?thesis
            proof (cases a)
              case (KValue x1)
              with 33(1) Some Pair Stackloc s1 show ?thesis using rexp.psimps(2) rexp_def by simp
            next
              case (KCDptr l')
              with 33 Some Pair Stackloc s1 show ?thesis using rexp.psimps(2)[of i r e cd st] rexp_def
by (simp split: option.split_asm Memoryvalue.split_asm MTypes.split_asm prod.split_asm Type.split_asm
StateMonad.result.split_asm)
            next
              case (KMemptr x3)
              with 33 Some Pair Stackloc s1 show ?thesis using rexp.psimps(2)[of i r e cd st] rexp_def
by (simp split: option.split_asm Memoryvalue.split_asm MTypes.split_asm prod.split_asm Type.split_asm
StateMonad.result.split_asm)
            next
              case (KStoptr x4)
              with 33 Some Pair Stackloc s1 show ?thesis using rexp.psimps(2)[of i r e cd
st] rexp_def by (simp split: option.split_asm STypes.split_asm prod.split_asm Type.split_asm
StateMonad.result.split_asm)
            qed
          qed
        next
          case (Storeloc x2)
          with 33 Some Pair show ?thesis using rexp.psimps rexp_def by (simp split: option.split_asm
STypes.split_asm prod.split_asm Type.split_asm  StateMonad.result.split_asm)
        qed
      qed
    qed
  qed
next
  case (34 e cd st g)
  then show ?case using expr.psimps(20) by (simp split:nat.split)
qed
```

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 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
  apply (relation "measures [mgas, msize]")
  apply (auto split:Member.split_asm prod.split_asm bool.split_asm if_split_asm Stackvalue.split_asm
option.split_asm Type.split_asm Types.split_asm Memoryvalue.split_asm atype.split_asm)
  apply (msel_ssel_expr_load_rexp_dom+, costs?, arith)+
  done
```

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))))"
by (induct rule: msel_ssel_expr_load_rexp.induct
[where ?P1.0="λc1 t1 l1 xe1 ev1 cd1 st1 g1. msel_ssel_expr_load_rexp_dom (Inl (Inl (c1, t1, l1, xe1,
ev1, cd1, st1, g1)))"
   and ?P2.0="λt2 l2 xe2 ev2 cd2 st2 g2. msel_ssel_expr_load_rexp_dom (Inl (Inr (t2, l2, xe2, ev2, cd2,
st2, g2)))"
   and ?P3.0="λe4 ev4 cd4 st4 g4. msel_ssel_expr_load_rexp_dom (Inr (Inl (e4, ev4, cd4, st4, g4)))"
   and ?P4.0="λlcp lis lxs lev0 lcd0 lk lm lev lcd lst lg. msel_ssel_expr_load_rexp_dom (Inr (Inr (Inl
(lcp, lis, lxs, lev0, lcd0, lk, lm, lev, lcd, lst, lg))))"
   and ?P5.0="λl3 ev3 cd3 st3 g3. msel_ssel_expr_load_rexp_dom (Inr (Inr (Inr (l3, ev3, cd3, st3,
g3))))"
], simp_all add: msel_ssel_expr_load_rexp.domintros)

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" using assms by (simp split:if_split_asm)

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 (induct rule: lexp.induct[where ?P="λl5 ev5 cd5 st5 g5. (∀l5' t5' g5'. lexp l5 ev5 cd5 st5 g5 =
Normal ((l5', t5'), g5') ⟶ g5' ≤ g5)"])
  case (1 i e uv st g)
  then show ?case using lexp.simps(1) by (simp split: option.split Denvalue.split prod.split)
next
  case (2 i r e cd st g)
  show ?case
  proof (rule allI[THEN allI, THEN allI, OF impI])
    fix st5' xa xaa
    assume a1: "lexp (Ref i r) e cd st g = Normal ((st5', xa), xaa)"
    then show "xaa ≤ g"
    proof (cases "fmlookup (denvalue e) i")
      case None
      with a1 show ?thesis using lexp.simps(2) by simp
    next
```

```
      case (Some a)
      then show ?thesis
      proof (cases a)
        case (Pair tp b)
        then show ?thesis
        proof (cases b)
          case (Stackloc l)
          then show ?thesis
          proof (cases "accessStore l (stack st)")
            case None
            with a1 Some Pair Stackloc show ?thesis using lexp.psimps(2) by simp
          next
            case s2: (Some a)
            then show ?thesis
            proof (cases a)
              case (KValue x1)
              with a1 Some Pair Stackloc s2 show ?thesis using lexp.psimps(2) by simp
            next
              case (KCDptr x2)
              with a1 Some Pair Stackloc s2 show ?thesis using lexp.psimps(2) by simp
            next
              case (KMemptr l')
              then show ?thesis
              proof (cases tp)
                case (Value _)
                with a1 Some Pair Stackloc s2 KMemptr show ?thesis using lexp.simps(2) by simp
              next
                case (Calldata _)
                with a1 Some Pair Stackloc s2 KMemptr show ?thesis using lexp.simps(2) by simp
              next
                case (Memory t)
                then show ?thesis
                proof (cases "msel True t l' r e cd st g")
                  case (n _ _)
                  with 2 a1 Some Pair Stackloc s2 KMemptr Memory show ?thesis using
msel_ssel_expr_load_rexp_gas(1) by (simp split: prod.split_asm)
                next
                  case (e _)
                  with a1 Some Pair Stackloc s2 KMemptr Memory show ?thesis using lexp.psimps(2) by
simp
                  qed
              next
                case (Storage _)
                with a1 Some Pair Stackloc s2 KMemptr show ?thesis using lexp.psimps(2) by simp
              qed
            next
              case (KStoptr l')
              then show ?thesis
              proof (cases tp)
                case (Value _)
                with a1 Some Pair Stackloc s2 KStoptr show ?thesis using lexp.psimps(2) by simp
              next
                case (Calldata _)
                with a1 Some Pair Stackloc s2 KStoptr show ?thesis using lexp.psimps(2) by simp
              next
                case (Memory _)
                with a1 Some Pair Stackloc s2 KStoptr show ?thesis using lexp.psimps(2) by simp
              next
                case (Storage t)
                then show ?thesis
                proof (cases "ssel t l' r e cd st g")
                  case (n _ _)
                  with a1 Some Pair Stackloc s2 KStoptr Storage show ?thesis using
msel_ssel_expr_load_rexp_gas(2) by (auto split: prod.split_asm)
```

```
                next
                  case (e _)
                  with a1 Some Pair Stackloc s2 KStoptr Storage show ?thesis using lexp.psimps(2) by
simp
                qed
              qed
            qed
          qed
        next
          case (Storeloc l)
          then show ?thesis
          proof (cases tp)
            case (Value _)
            with a1 Some Pair Storeloc show ?thesis using lexp.psimps(2) by simp
          next
            case (Calldata _)
            with  a1 Some Pair Storeloc show ?thesis using lexp.psimps(2) by simp
          next
            case (Memory _)
            with a1 Some Pair Storeloc show ?thesis using lexp.psimps(2) by simp
          next
            case (Storage t)
            then show ?thesis
            proof (cases "ssel t l r e cd st g")
              case (n _ _)
              with a1 Some Pair Storeloc Storage show ?thesis using msel_ssel_expr_load_rexp_gas(2)
by (auto split: prod.split_asm)
            next
              case (e _)
              with a1 Some Pair Storeloc Storage show ?thesis using lexp.psimps(2) by simp
            qed
          qed
        qed
      qed
    qed
  qed
qed
```

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"
  using assms unfolding wp_def by (simp split:result.split result.split_asm)
```

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"
by pat_completeness auto
```

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 (induct rule: stmt.pinduct[where ?P="λs6 ev6 cd6 st6. (∀st6'. stmt s6 ev6 cd6 st6 = Normal ((),
st6') ⟶ gas st6' ≤ gas st6)"])
  case (1 e cd st)
  then show ?case using stmt.psimps(1) by simp
next
  case (2 lv ex env cd st)
  define g where "g = costs (ASSIGN lv ex) env cd st"
  show ?case
  proof (rule allI[OF impI])
    fix st6'
    assume stmt_def: "stmt (ASSIGN lv ex) env cd st = Normal ((), st6')"
    then show "gas st6' ≤ gas st"
    proof cases
      assume "gas st ≤ g"
      with 2(1) stmt_def show ?thesis using stmt.psimps(2) g_def by simp
    next
      assume "¬ gas st ≤ g"
      define st' where "st' = st⦇gas := gas st - g⦈"
      show ?thesis
      proof (cases "expr ex env cd st' (gas st - g)")
        case (n a g')
        define st'' where "st'' = st'⦇gas := g'⦈"
        then show ?thesis
        proof (cases a)
          case (Pair b c)
          then show ?thesis
          proof (cases b)
            case (KValue v)
            then show ?thesis
            proof (cases c)
              case (Value t)
              then show ?thesis
              proof (cases "lexp lv env cd st'' g'")
                case n2: (n a g'')
                then show ?thesis
                proof (cases a)
                  case p1: (Pair a b)
                  then show ?thesis
                  proof (cases a)
                    case (LStackloc l)
                    then show ?thesis
                    proof (cases b)
                      case v2: (Value t')
                      then show ?thesis
                      proof (cases "convert t t' v ")
                        case None
                        with stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p1 LStackloc v2 show
?thesis using stmt.psimps(2)[OF 2(1)] g_def st'_def st''_def by simp
                      next
                        case s3: (Some v')
                        with 2(1) ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p1 LStackloc v2 s3
                        have "stmt (ASSIGN lv ex) env cd st = Normal ((), st''⦇gas := g'', stack :=
updateStore l (KValue v') (stack st)⦈)"
```

```
                    using stmt.psimps(2) g_def st'_def st''_def by simp
                   with stmt_def have "st6'= st''⦇gas := g'', stack := updateStore l (KValue v')
(stack st)⦈" by simp
                   moreover from lexp_gas `¬ gas st ≤ g` n2 p1 have "gas (st''⦇gas := g'',
stack := updateStore l (KValue v') (stack st)⦈) ≤ gas (st'⦇gas := g'⦈)" using g_def st'_def by simp
                   moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st' "gas st - g"]
`¬ gas st ≤ g` n Pair KValue Value have "gas (st'⦇gas := g'⦈) ≤ gas (st⦇gas := gas st - g⦈)" using
g_def by simp
                    ultimately show ?thesis by simp
                  qed
                 next
                  case (Calldata x2)
                  with 2(1) stmt_def `¬ gas st ≤ g` n Pair KValue Value n2 p1 LStackloc show
?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                 next
                  case (Memory x3)
                  with 2(1) stmt_def `¬ gas st ≤ g` n Pair KValue Value n2 p1 LStackloc show
?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                 next
                  case (Storage x4)
                  with 2(1) stmt_def `¬ gas st ≤ g` n Pair KValue Value n2 p1 LStackloc show
?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                 qed
                next
                  case (LMemloc l)
                  then show ?thesis
                  proof (cases b)
                    case v2: (Value t')
                    with 2(1) stmt_def `¬ gas st ≤ g` n Pair KValue Value n2 p1 LMemloc show
?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                  next
                    case (Calldata x2)
                    with 2(1) stmt_def `¬ gas st ≤ g` n Pair KValue Value n2 p1 LMemloc show
?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                  next
                    case (Memory x3)
                    then show ?thesis
                    proof (cases x3)
                      case (MTArray x11 x12)
                      with 2(1) stmt_def `¬ gas st ≤ g` n Pair KValue Value n2 p1 LMemloc Memory
show ?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                     next
                      case (MTValue t')
                      then show ?thesis
                      proof (cases "convert t t' v ")
                        case None
                        with 2(1) stmt_def `¬ gas st ≤ g` n Pair KValue Value n2 p1 LMemloc Memory
MTValue show ?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                      next
                        case s3: (Some v')
                        with 2(1) `¬ gas st ≤ g` n Pair KValue Value n2 p1 LMemloc Memory MTValue s3
                        have "stmt (ASSIGN lv ex) env cd st = Normal ((), st''⦇gas := g'', memory :=
updateStore l (MValue v') (memory st'')⦈)"
                          using stmt.psimps(2) g_def st'_def st''_def by simp
                        with stmt_def have "st6'= (st''⦇gas := g'', memory := updateStore l (MValue
v') (memory st'')⦈)" by simp
                        moreover from lexp_gas `¬ gas st ≤ g` n2 p1 have "gas (st''⦇gas := g'',
stack := updateStore l (KValue v') (stack st)⦈) ≤ gas (st'⦇gas := g'⦈)" using g_def st'_def by simp
                        moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st' "gas st - g"]
`¬ gas st ≤ g` n Pair KValue Value n2 p1 have "gas (st'⦇gas := g'⦈) ≤ gas (st⦇gas := gas st - g⦈)"
using g_def by simp
                        ultimately show ?thesis by simp
                      qed
                     qed
```

```
                    next
                      case (Storage x4)
                      with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p1 LMemloc show
?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                    qed
                  next
                    case (LStoreloc l)
                    then show ?thesis
                    proof (cases b)
                      case v2: (Value t')
                      with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p1 LStoreloc show
?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                    next
                      case (Calldata x2)
                      with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p1 LStoreloc show
?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                    next
                      case (Memory x3)
                      with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p1 LStoreloc show
?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                    next
                      case (Storage x4)
                      then show ?thesis
                      proof (cases x4)
                        case (STArray x11 x12)
                        with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p1 LStoreloc Storage
show ?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                      next
                        case (STMap x21 x22)
                        with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p1 LStoreloc Storage
show ?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                      next
                        case (STValue t')
                        then show ?thesis
                        proof (cases "convert t t' v ")
                          case None
                          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p1 LStoreloc
Storage STValue show ?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                        next
                          case s3: (Some v')
                          with 2(1) ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p1 LStoreloc Storage STValue
s3
                          have "stmt (ASSIGN lv ex) env cd st = Normal ((), st'' ⦇gas := g'', storage
:= (storage st'') (address env := fmupd l v' (storage st'' (address env)))⦈)"
                            using stmt.psimps(2) g_def st'_def st''_def by simp
                          with stmt_def have "st6'= st'' ⦇gas := g'', storage := (storage st'')
(address env := fmupd l v' (storage st'' (address env)))⦈" by simp
                          moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p1 have "gas
(st''⦇gas := g'', stack := updateStore l (KValue v') (stack st)⦈) ≤ gas (st'⦇gas := g'⦈)" using g_def
by simp
                          moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st' "gas st - g"]
‘¬ gas st ≤ g‘ n Pair KValue Value n2 p1 have "gas (st'⦇gas := g'⦈) ≤ gas (st⦇gas := gas st - g⦈)"
using g_def by simp
                          ultimately show ?thesis by simp
                        qed
                      qed
                    qed
                  qed
                qed
              next
                case (e x)
                with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value show ?thesis using
stmt.psimps(2) g_def st'_def st''_def by simp
              qed
```

```
            next
              case (Calldata x2)
              with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue show ?thesis using stmt.psimps(2) g_def
st'_def by simp
            next
              case (Memory x3)
              with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue show ?thesis using stmt.psimps(2) g_def
st'_def by simp
            next
              case (Storage x4)
              with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue show ?thesis using stmt.psimps(2) g_def
st'_def by simp
            qed
          next
            case (KCDptr p)
            then show ?thesis
            proof (cases c)
              case (Value x1)
              with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr show ?thesis using stmt.psimps(2) g_def
st'_def by simp
            next
              case (Calldata x2)
              then show ?thesis
              proof (cases x2)
                case (MTArray x t)
                then show ?thesis
                proof (cases "lexp lv env cd st'' g'")
                  case n2: (n a g'')
                  define st''' where "st''' = st''⦇gas := g''⦈"
                  then show ?thesis
                  proof (cases a)
                    case p2: (Pair a b)
                    then show ?thesis
                    proof (cases a)
                      case (LStackloc l)
                      then show ?thesis
                      proof (cases b)
                        case v2: (Value t')
                        with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
LStackloc show ?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                      next
                        case c2: (Calldata x2)
                        with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
LStackloc show ?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                      next
                        case (Memory x3)
                        then show ?thesis
                        proof (cases "accessStore l (stack st''')")
                          case None
                          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
LStackloc Memory show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                        next
                          case s3: (Some a)
                          then show ?thesis
                          proof (cases a)
                            case (KValue x1)
                            with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
LStackloc Memory s3 show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                          next
                            case c3: (KCDptr x2)
                            with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
LStackloc Memory s3 show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                          next
                            case (KMemptr p')
```

```
                        then show ?thesis
                        proof (cases "cpm2m p p' x t cd (memory st''')")
                          case None
                          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
LStackloc Memory s3 KMemptr show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by
(simp split:if_split_asm)
                        next
                          case (Some m')
                          with ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2 LStackloc
Memory s3 KMemptr
                          have "stmt (ASSIGN lv ex) env cd st = Normal ((), st''' (|memory := m'|))"
                            using stmt.psimps(2)[OF 2(1)] g_def st'_def st''_def st'''_def by simp
                          with stmt_def have "st6'= st''' (|memory := m'|)" by simp
                          moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2
p2 have "gas (st'''(|memory := m'|)) ≤ gas st''" using st''_def st'''_def by simp
                          moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st' "gas st -
g"] ‘¬ gas st ≤ g‘ n Pair have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                          ultimately show ?thesis using st'_def by simp
                        qed
                      next
                        case (KStoptr p')
                        with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
LStackloc Memory s3 show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                      qed
                    qed
                  next
                    case (Storage x4)
                    then show ?thesis
                    proof (cases "accessStore l (stack st'')")
                      case None
                      with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
LStackloc Storage show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                    next
                      case s3: (Some a)
                      then show ?thesis
                      proof (cases a)
                        case (KValue x1)
                        with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
LStackloc Storage s3 show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                      next
                        case c3: (KCDptr x2)
                        with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
LStackloc Storage s3 show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                      next
                        case (KMemptr x3)
                        with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
LStackloc Storage s3 show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                      next
                        case (KStoptr p')
                        then show ?thesis
                        proof (cases "cpm2s p p' x t cd (storage st'' (address env))")
                          case None
                          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
LStackloc Storage s3 KStoptr show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by
simp
                        next
                          case (Some s')
                          with 2(1) ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2 LStackloc
Storage s3 KStoptr
                          have "stmt (ASSIGN lv ex) env cd st = Normal ((), st''' (|storage :=
(storage st'') (address env := s')|))"
                            using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                          with stmt_def have "st6'= st''' (|storage := (storage st'') (address env
:= s')|)" by simp
```

```
                            moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2
p2 have "gas st’’’ ≤ gas st’’" using g_def st’’_def st’’’_def by simp
                            moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st’ "gas st -
g"] ‘¬ gas st ≤ g‘ n Pair have "gas st’’ ≤ gas st’" using g_def st’_def st’’_def by simp
                            ultimately show ?thesis using st’_def by simp
                          qed
                        qed
                      qed
                    qed
                  next
                    case (LMemloc l)
                    then show ?thesis
                    proof (cases "cpm2m p l x t cd (memory st’’’)")
                      case None
                      with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2 LMemloc
show ?thesis using stmt.psimps(2) g_def st’_def st’’_def st’’’_def by (simp split:if_split_asm)
                    next
                      case (Some m)
                      with ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2 LMemloc
                      have "stmt (ASSIGN lv ex) env cd st = Normal ((), st’’’(|memory := m|))"
                        using stmt.psimps(2)[OF 2(1)] g_def st’_def st’’_def st’’’_def by simp
                      with stmt_def have "st6’= (st’’’(|memory := m|))" by simp
                      moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
have "gas st’’’ ≤ gas st’’" using st’’_def st’’’_def by simp
                      moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st’ "gas st - g"]
‘¬ gas st ≤ g‘ n Pair have "gas st’’ ≤ gas st’" using st’_def st’’_def by simp
                      ultimately show ?thesis using st’_def by simp
                    qed
                  next
                    case (LStoreloc l)
                    then show ?thesis
                    proof (cases "cpm2s p l x t cd (storage st’’ (address env))")
                      case None
                      with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
LStoreloc show ?thesis using stmt.psimps(2) g_def st’_def st’’_def st’’’_def by simp
                    next
                      case (Some s)
                      with ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2 LStoreloc
                      have "stmt (ASSIGN lv ex) env cd st = Normal ((), st’’’ (|storage := (storage
st’’) (address env := s)|))"
                        using stmt.psimps(2)[OF 2(1)] g_def st’_def st’’_def st’’’_def by simp
                      with stmt_def have "st6’= (st’’’(|storage := (storage st’’) (address env :=
s)|))" by simp
                      moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray n2 p2
have "gas st’’’ ≤ gas st’’" using st’’_def st’’’_def by simp
                      moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st’ "gas st - g"]
‘¬ gas st ≤ g‘ n Pair have "gas st’’ ≤ gas st’" using st’_def st’’_def by simp
                      ultimately show ?thesis using st’_def by simp
                    qed
                  qed
                qed
              next
                case (e x)
                with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata MTArray show ?thesis using
stmt.psimps(2) g_def st’_def st’’_def by simp
              qed
            next
              case (MTValue x2)
              with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr Calldata show ?thesis using
stmt.psimps(2) g_def st’_def st’’_def by simp
            qed
          next
            case (Memory x3)
            with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr show ?thesis using stmt.psimps(2) g_def
```

```
st'_def st''_def by simp
            next
              case (Storage x4)
              with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KCDptr show ?thesis using stmt.psimps(2) g_def
st'_def st''_def by simp
              qed
           next
             case (KMemptr p)
             then show ?thesis
             proof (cases c)
               case (Value x1)
               with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KMemptr show ?thesis using stmt.psimps(2)
g_def st'_def st''_def by simp
             next
               case (Calldata x2)
               with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KMemptr show ?thesis using stmt.psimps(2)
g_def st'_def st''_def by simp
             next
               case (Memory x3)
               then show ?thesis
               proof (cases x3)
                 case (MTArray x t)
                 then show ?thesis
                 proof (cases "lexp lv env cd st'' g'")
                   case n2: (n a g'')
                   define st''' where "st''' = st''⦇gas := g''⦈"
                   then show ?thesis
                   proof (cases a)
                     case p2: (Pair a b)
                     then show ?thesis
                     proof (cases a)
                       case (LStackloc l)
                       then show ?thesis
                       proof (cases b)
                         case v2: (Value t')
                         with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2
LStackloc show ?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                       next
                         case c2: (Calldata x2)
                         with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2
LStackloc show ?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                       next
                         case m2: (Memory x3)
                         with 2(1) ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2 LStackloc
                         have "stmt (ASSIGN lv ex) env cd st = Normal ((), st'''⦇stack := updateStore l
(KMemptr p) (stack st'')⦈)"
                           using stmt.psimps(2)[OF 2(1)] g_def st'_def st''_def st'''_def by simp
                         with stmt_def have "st6'= st'''⦇stack := updateStore l (KMemptr p) (stack
st'')⦈" by simp
                         moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2
have "gas st''' ≤ gas st'''" using st''_def st'''_def by simp
                         moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st' "gas st - g"]
‘¬ gas st ≤ g‘ n Pair have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                         ultimately show ?thesis using st'_def by simp
                       next
                         case (Storage x4)
                         then show ?thesis
                         proof (cases "accessStore l (stack st''')")
                           case None
                           with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2
LStackloc Storage show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                         next
                           case s3: (Some a)
                           then show ?thesis
```

```
                        proof (cases a)
                          case (KValue x1)
                          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2
LStackloc Storage s3 show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                        next
                          case c3: (KCDptr x2)
                          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2
LStackloc Storage s3 show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                        next
                          case m3: (KMemptr x3)
                          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2
LStackloc Storage s3 show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                        next
                          case (KStoptr p')
                          then show ?thesis
                          proof (cases "cpm2s p p' x t (memory st''') (storage st''' (address env))")
                            case None
                            with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2
LStackloc Storage s3 KStoptr show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by
simp
                          next
                            case (Some s)
                            with 2(1) ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2 LStackloc
Storage s3 KStoptr
                            have "stmt (ASSIGN lv ex) env cd st = Normal ((), st'''⦇storage :=
(storage st''') (address env := s)⦈)"
                              using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                            with stmt_def have "st6'= st'''⦇storage := (storage st''') (address env
:= s)⦈" by simp
                            moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2
p2 have "gas st''' ≤ gas st''" using g_def st'_def st''_def st'''_def by simp
                            moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st' "gas st -
g"] ‘¬ gas st ≤ g‘ n Pair have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                            ultimately show ?thesis using st'_def by simp
                          qed
                        qed
                      qed
                    qed
                  next
                    case (LMemloc l)
                    with 2(1) ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2 LMemloc
                    have "stmt (ASSIGN lv ex) env cd st =  Normal ((), st'''⦇memory := updateStore l
(MPointer p) (memory st''')⦈)"
                      using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                    with stmt_def have "st6'= st'''⦇memory := updateStore l (MPointer p) (memory
st''')⦈" by simp
                    moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2 have
"gas st''' ≤ gas st''" using g_def st'_def st''_def st'''_def by simp
                    moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st' "gas st - g"] ‘¬
gas st ≤ g‘ n Pair have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                    ultimately show ?thesis using st'_def by simp
                  next
                    case (LStoreloc l)
                    then show ?thesis
                    proof (cases "cpm2s p l x t (memory st''') (storage st'' (address env))")
                      case None
                      with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2
LStoreloc show ?thesis using stmt.psimps(2) g_def using st'_def st''_def st'''_def by simp
                    next
                      case (Some s)
                      with 2(1) ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2 LStoreloc
                      have "stmt (ASSIGN lv ex) env cd st =  Normal ((), st'''⦇storage := (storage
st''') (address env := s)⦈)"
                        using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
```

```
                    with stmt_def have "st6'= st'''⦇storage := (storage st''') (address env := s)⦈"
by simp
                    moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray n2 p2
have "gas st''' ≤ gas st''" using g_def st'_def st''_def st'''_def by simp
                    moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st' "gas st - g"]
‘¬ gas st ≤ g‘ n Pair have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                    ultimately show ?thesis using st'_def by simp
                  qed
                qed
              qed
            next
              case (e _)
              with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KMemptr Memory MTArray show ?thesis using
stmt.psimps(2) g_def st'_def st''_def by simp
            qed
          next
            case (MTValue _)
            with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KMemptr Memory show ?thesis using
stmt.psimps(2) g_def st'_def st''_def by simp
          qed
        next
          case (Storage x4)
          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KMemptr show ?thesis using stmt.psimps(2)
g_def st'_def st''_def by simp
        qed
      next
        case (KStoptr p)
        then show ?thesis
        proof (cases c)
          case (Value x1)
          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr show ?thesis using stmt.psimps(2)
g_def st'_def st''_def by simp
        next
          case (Calldata x2)
          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr show ?thesis using stmt.psimps(2)
g_def st'_def st''_def by simp
        next
          case (Memory x3)
          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr show ?thesis using stmt.psimps(2)
g_def st'_def st''_def by simp
        next
          case (Storage x4)
          then show ?thesis
          proof (cases x4)
            case (STArray x t)
            then show ?thesis
            proof (cases "lexp lv env cd st'' g'")
              case n2: (n a g'')
              define st''' where "st''' = st''⦇gas := g''⦈"
              then show ?thesis
              proof (cases a)
                case p2: (Pair a b)
                then show ?thesis
                proof (cases a)
                  case (LStackloc l)
                  then show ?thesis
                  proof (cases b)
                    case v2: (Value t')
                    with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2
LStackloc show ?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                  next
                    case c2: (Calldata x2)
                    with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2
LStackloc show ?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
```

```
                    next
                      case (Memory x3)
                      then show ?thesis
                      proof (cases "accessStore l (stack st''')")
                        case None
                        with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2
LStackloc Memory show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                      next
                        case s3: (Some a)
                        then show ?thesis
                        proof (cases a)
                          case (KValue x1)
                          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2
LStackloc Memory s3 show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                        next
                          case c3: (KCDptr x2)
                          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2
LStackloc Memory s3 show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                        next
                          case (KMemptr p')
                          then show ?thesis
                          proof (cases "cps2m p p' x t (storage st''' (address env)) (memory st''')")
                            case None
                            with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2
LStackloc Memory s3 KMemptr show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                          next
                            case (Some m)
                            with 2(1) ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2 LStackloc
Memory s3 KMemptr
                            have "stmt (ASSIGN lv ex) env cd st =  Normal ((), st'''⟦memory := m⟧)"
                              using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                            with stmt_def have "st6'= st'''⟦memory := m⟧" by simp
                            moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KMemptr Storage STArray n2
p2 have "gas (st'''⟦memory := m⟧) ≤ gas st'''" using g_def st'_def st''_def st'''_def by simp
                            moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st' "gas st -
g"] ‘¬ gas st ≤ g‘ n Pair have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                            ultimately show ?thesis using st'_def by simp
                          qed
                        next
                          case sp2: (KStoptr p')
                          with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2
LStackloc Memory s3 show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                          qed
                      qed
                    next
                      case st2: (Storage x4)
                      with 2(1) ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2 LStackloc
                      have "stmt (ASSIGN lv ex) env cd st =  Normal ((), st'''⟦stack := updateStore l
(KStoptr p) (stack st''')⟧)"
                        using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                      with stmt_def have "st6'= st'''⟦stack := updateStore l (KStoptr p) (stack
st''')⟧" by simp
                      moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2
have "gas (st'''⟦stack := updateStore l (KStoptr p) (stack st''')⟧) ≤ gas st'''" using g_def st'_def
st''_def st'''_def by simp
                      moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st' "gas st - g"]
‘¬ gas st ≤ g‘ n Pair have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                      ultimately show ?thesis using st'_def by simp
                    qed
                  next
                    case (LMemloc l)
                    then show ?thesis
                    proof (cases "cps2m p l x t (storage st'' (address env)) (memory st'')")
                      case None
```

```
                    with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2 LMemloc
show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                  next
                    case (Some m)
                    with 2(1) ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2 LMemloc
                    have "stmt (ASSIGN lv ex) env cd st =  Normal ((), st'''⦇memory := m⦈)"
                      using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                    with stmt_def have "st6'= (st'''⦇memory := m⦈)" by simp
                    moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2
have "gas (st'''⦇memory := m⦈) ≤ gas st'''" using g_def st'_def st''_def st'''_def by simp
                    moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st' "gas st - g"]
‘¬ gas st ≤ g‘ n Pair have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                    ultimately show ?thesis using st'_def by simp
                  qed
                next
                  case (LStoreloc l)
                  then show ?thesis
                  proof (cases "copy p l x t (storage st'' (address env))")
                    case None
                    with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2
LStoreloc show ?thesis using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                  next
                    case (Some s)
                    with 2(1) ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2 LStoreloc
                    have "stmt (ASSIGN lv ex) env cd st = Normal ((), st'''⦇storage := (storage
st''') (address env := s)⦈)"
                      using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                    with stmt_def have "st6'= st'''⦇storage := (storage st''') (address env := s)⦈"
by simp
                    moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray n2 p2
have "gas (st'''⦇storage := (storage st''') (address env := s')⦈) ≤ gas st'''" using g_def st'_def
st''_def st'''_def by simp
                    moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st' "gas st - g"]
‘¬ gas st ≤ g‘ n Pair have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                    ultimately show ?thesis using st'_def by simp
                  qed
                qed
              qed
            next
              case (e x)
              with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STArray show ?thesis using
stmt.psimps(2) g_def st'_def st''_def by simp
            qed
          next
            case (STMap t t')
            then show ?thesis
            proof (cases "lexp lv env cd st'' g'")
              case n2: (n a g'')
              define st''' where "st''' = st''⦇gas := g''⦈"
              then show ?thesis
              proof (cases a)
                case p2: (Pair a b)
                then show ?thesis
                proof (cases a)
                  case (LStackloc l)
                  with 2(1) ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STMap n2 p2
                  have "stmt (ASSIGN lv ex) env cd st = Normal ((), st''' ⦇stack := updateStore l
(KStoptr p) (stack st''')⦈)"
                    using stmt.psimps(2) g_def st'_def st''_def st'''_def by simp
                  with stmt_def have "st6'= st'''⦇stack := updateStore l (KStoptr p) (stack
st''')⦈" by simp
                  moreover from lexp_gas ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STMap n2 p2 have
"gas (st'''⦇stack := updateStore l (KStoptr p) (stack st''')⦈) ≤ gas st'''" using g_def st'_def st''_def
st'''_def by simp
```

```
                    moreover from msel_ssel_expr_load_rexp_gas(3)[of ex env cd st' "gas st - g"] ‘¬
gas st ≤ g‘ n Pair have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                    ultimately show ?thesis using st'_def by simp
                  next
                    case (LMemloc x2)
                    with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STMap n2 p2 show
?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                  next
                    case (LStoreloc x3)
                    with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STMap n2 p2 show
?thesis using stmt.psimps(2) g_def st'_def st''_def by simp
                  qed
                qed
              next
                case (e x)
                with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage STMap show ?thesis using
stmt.psimps(2) g_def st'_def st''_def by simp
              qed
            next
              case (STValue x3)
              with 2(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KStoptr Storage show ?thesis using
stmt.psimps(2) g_def st'_def st''_def by simp
            qed
          qed
        qed
      qed
    next
      case (e x)
      with 2(1) stmt_def ‘¬ gas st ≤ g‘ show ?thesis using stmt.psimps(2) g_def st'_def by simp
    qed
  qed
qed
next
  case (3 s1 s2 e cd st)
  define g where "g = costs (COMP s1 s2) e cd st"
  show ?case
  proof (rule allI[OF impI])
    fix st6'
    assume stmt_def: "stmt (COMP s1 s2) e cd st = Normal ((), st6')"
    then show "gas st6' ≤ gas st"
    proof cases
      assume "gas st ≤ g"
      with 3(1) stmt_def g_def show ?thesis using stmt.psimps(3) by simp
    next
      assume "¬ gas st ≤ g"
      show ?thesis
      proof (cases "stmt s1 e cd (st⦇gas := gas st - g⦈)")
        case (n a st')
        with 3(1) stmt_def ‘¬ gas st ≤ g‘ have "stmt (COMP s1 s2) e cd st = stmt s2 e cd st'" using
stmt.psimps(3)[of s1 s2 e cd st] g_def by (simp add:Let_def split:unit.split_asm)
        with 3(3) stmt_def ‹¬ gas st ≤ g› n have "gas st6' ≤ gas st'" using g_def by simp
        moreover from 3(2)[where ?s'a="st⦇gas := gas st - g⦈"] ‹¬ gas st ≤ g› n have "gas st' ≤
gas st" using g_def by simp
        ultimately show ?thesis by simp
      next
        case (e x)
        with 3 stmt_def ‘¬ gas st ≤ g‘ show ?thesis using stmt.psimps(3)[of s1 s2 e cd st] g_def by
(simp split: Ex.split_asm)
      qed
    qed
  qed
next
  case (4 ex s1 s2 e cd st)
  define g where "g = costs (ITE ex s1 s2) e cd st"
```

```
  show ?case
  proof (rule allI[OF impI])
    fix st6'
    assume stmt_def: "stmt (ITE ex s1 s2) e cd st = Normal ((), st6')"
    then show "gas st6' ≤ gas st"
    proof cases
      assume "gas st ≤ g"
      with 4(1) stmt_def show ?thesis using stmt.psimps(4) g_def by simp
    next
      assume "¬ gas st ≤ g"
      then have l1: "assert Gas (λst. costs (ITE ex s1 s2) e cd st < gas st) st = Normal ((), st) "
using g_def by simp
      define st' where "st' = st(|gas := gas st - g|)"
      then have l2: " modify (λst. st(|gas := gas st - costs (ITE ex s1 s2) e cd st|)) st = Normal ((),
st')" using g_def by simp
      show ?thesis
      proof (cases "expr ex e cd st' (gas st - g)")
        case (n a g')
        define st'' where "st'' = st'(|gas := g'|)"
        with n have l3: "toState (expr ex e cd) st' = Normal (a, st'')" using st'_def by simp
        then show ?thesis
        proof (cases a)
          case (Pair b c)
          then show ?thesis
          proof (cases b)
            case (KValue b)
            then show ?thesis
            proof (cases c)
              case (Value x1)
              then show ?thesis
              proof (cases x1)
                case (TSInt x1)
                with 4(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value show ?thesis using
stmt.psimps(4) g_def st'_def by simp
              next
                case (TUInt x2)
                with 4(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value show ?thesis using
stmt.psimps(4) g_def st'_def by simp
              next
                case TBool
                then show ?thesis
                proof cases
                  assume "b = ShowL_bool True"
                  with 4(1) ‘¬ gas st ≤ g‘ n Pair KValue Value TBool have "stmt (ITE ex s1 s2) e cd
st = stmt s1 e cd st''" using stmt.psimps(4) g_def st'_def st''_def by simp
                  with 4(2)[OF l1 l2 l3] stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value TBool ‘b =
ShowL_bool True‘ have "gas st6' ≤ gas st''" using g_def by simp
                  moreover from msel_ssel_expr_load_rexp_gas(3)[of ex e cd st' "gas st - g"] ‘¬ gas
st ≤ g‘ n Pair KValue Value TBool have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                  ultimately show ?thesis using st'_def by simp
                next
                  assume nt: "¬ b = ShowL_bool True"
                  show ?thesis
                  proof cases
                    assume "b = ShowL_bool False"
                    with 4(1) ‘¬ gas st ≤ g‘ n Pair KValue Value TBool nt have "stmt (ITE ex s1 s2) e
cd st = stmt s2 e cd st''" using stmt.psimps(4) g_def st'_def st''_def by simp
                    with 4(3)[OF l1 l2 l3] stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value TBool nt ‘b =
ShowL_bool False‘ have "gas st6' ≤ gas st''" using g_def by simp
                    moreover from msel_ssel_expr_load_rexp_gas(3)[of ex e cd st' "gas st - g"] ‘¬ gas
st ≤ g‘ n Pair KValue Value TBool have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                    ultimately show ?thesis using st'_def by simp
                  next
                    assume "¬ b = ShowL_bool False"
```

```
                with 4(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value TBool nt show ?thesis using
stmt.psimps(4) g_def st'_def st''_def by simp
              qed
            qed
          next
            case TAddr
            with 4(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value show ?thesis using
stmt.psimps(4) g_def st'_def st''_def by simp
          qed
        next
          case (Calldata x2)
          with 4(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue show ?thesis using stmt.psimps(4) g_def
st'_def st''_def by simp
        next
          case (Memory x3)
          with 4(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue show ?thesis using stmt.psimps(4) g_def
st'_def st''_def by simp
        next
          case (Storage x4)
          with 4(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue show ?thesis using stmt.psimps(4) g_def
st'_def st''_def by simp
        qed
      next
        case (KCDptr x2)
        with 4(1) stmt_def ‘¬ gas st ≤ g‘ n Pair show ?thesis using stmt.psimps(4) g_def st'_def
st''_def by simp
      next
        case (KMemptr x3)
        with 4(1) stmt_def ‘¬ gas st ≤ g‘ n Pair show ?thesis using stmt.psimps(4) g_def st'_def
st''_def by simp
      next
        case (KStoptr x4)
        with 4(1) stmt_def ‘¬ gas st ≤ g‘ n Pair show ?thesis using stmt.psimps(4) g_def st'_def
st''_def by simp
      qed
    qed
  next
    case (e e)
    with 4(1) stmt_def ‘¬ gas st ≤ g‘ show ?thesis using stmt.psimps(4) g_def st'_def by simp
  qed
 qed
qed
next
 case (5 ex s0 e cd st)
 define g where "g = costs (WHILE ex s0) e cd st"
 show ?case
 proof (rule allI[OF impI])
  fix st6'
  assume stmt_def: "stmt (WHILE ex s0) e cd st = Normal ((), st6')"
  then show "gas st6' ≤ gas st"
  proof cases
    assume "gas st ≤ g"
    with 5(1) stmt_def show ?thesis using stmt.psimps(5) g_def by simp
  next
    assume gcost: "¬ gas st ≤ g"
    then have l1: "assert Gas (λst. costs (WHILE ex s0) e cd st < gas st) st = Normal ((), st) "
using g_def by simp
    define st' where "st' = st(|gas := gas st - g|)"
    then have l2: " modify (λst. st(|gas := gas st - costs (WHILE ex s0) e cd st|)) st = Normal ((),
st')" using g_def by simp
    show ?thesis
    proof (cases "expr ex e cd st' (gas st - g)")
      case (n a g')
      define st'' where "st'' = st'(|gas := g'|)"
```

```
        with n have l3: "toState (expr ex e cd) st' = Normal (a, st'')" using st'_def by simp
        then show ?thesis
        proof (cases a)
          case (Pair b c)
          then show ?thesis
          proof (cases b)
            case (KValue b)
            then show ?thesis
            proof (cases c)
              case (Value x1)
              then show ?thesis
              proof (cases x1)
                case (TSInt x1)
                with 5(1) stmt_def gcost n Pair KValue Value show ?thesis using stmt.psimps(5) g_def
st'_def by simp
              next
                case (TUInt x2)
                with 5(1) stmt_def gcost n Pair KValue Value show ?thesis using stmt.psimps(5) g_def
st'_def by simp
              next
                case TBool
                then show ?thesis
                proof cases
                  assume "b = ShowL_bool True"
                  then show ?thesis
                  proof (cases "stmt s0 e cd st''")
                    case n2: (n a st''')
                    with 5(1) gcost n Pair KValue Value TBool `b = ShowL_bool True` have "stmt (WHILE
ex s0) e cd st = stmt (WHILE ex s0) e cd st'''" using stmt.psimps(5)[of ex s0 e cd st] g_def st'_def
st''_def by (simp add: Let_def split:unit.split_asm)
                    with 5(3) stmt_def gcost n2 Pair KValue Value TBool `b = ShowL_bool True` n have
"gas st6' ≤ gas st'''" using g_def st'_def st''_def by simp
                    moreover from 5(2)[OF l1 l2 l3] gcost n2 Pair KValue Value TBool `b = ShowL_bool
True` n have "gas st''' ≤ gas st''" using g_def by simp
                    moreover from msel_ssel_expr_load_rexp_gas(3)[of ex e cd st' "gas st - g"] `¬ gas
st ≤ g` n Pair KValue Value TBool have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                    ultimately show ?thesis using st'_def by simp
                  next
                    case (e x)
                    with 5(1) stmt_def gcost n Pair KValue Value TBool `b = ShowL_bool True` show
?thesis using stmt.psimps(5) g_def st'_def st''_def by simp
                  qed
                next
                  assume nt: "¬ b = ShowL_bool True"
                  show ?thesis
                  proof cases
                    assume "b = ShowL_bool False"
                    with 5(1) gcost n Pair KValue Value TBool nt have "stmt (WHILE ex s0) e cd st =
return () st''" using stmt.psimps(5) g_def st'_def st''_def by simp
                    with stmt_def have "gas st6' ≤ gas st''" by simp
                    moreover from msel_ssel_expr_load_rexp_gas(3)[of ex e cd st' "gas st - g"] `¬ gas
st ≤ g` n Pair KValue Value TBool have "gas st'' ≤ gas st'" using st'_def st''_def by simp
                    ultimately show ?thesis using g_def st'_def st''_def by simp
                  next
                    assume "¬ b = ShowL_bool False"
                    with 5(1) stmt_def gcost n Pair KValue Value TBool nt show ?thesis using
stmt.psimps(5) g_def st'_def st''_def by simp
                  qed
                qed
              next
                case TAddr
                with 5(1) stmt_def gcost n Pair KValue Value show ?thesis using stmt.psimps(5) g_def
st'_def st''_def by simp
              qed
```

```
            next
              case (Calldata x2)
              with 5(1) stmt_def gcost n Pair KValue show ?thesis using stmt.psimps(5) g_def st'_def
st''_def by simp
            next
              case (Memory x3)
              with 5(1) stmt_def gcost n Pair KValue show ?thesis using stmt.psimps(5) g_def st'_def
st''_def by simp
            next
              case (Storage x4)
              with 5(1) stmt_def gcost n Pair KValue show ?thesis using stmt.psimps(5) g_def st'_def
st''_def by simp
            qed
          next
            case (KCDptr x2)
            with 5(1) stmt_def gcost n Pair show ?thesis using stmt.psimps(5) g_def st'_def st''_def
by simp
          next
            case (KMemptr x3)
            with 5(1) stmt_def gcost n Pair show ?thesis using stmt.psimps(5) g_def st'_def st''_def
by simp
          next
            case (KStoptr x4)
            with 5(1) stmt_def gcost n Pair show ?thesis using stmt.psimps(5) g_def st'_def st''_def
by simp
          qed
        qed
      next
        case (e e)
        with 5(1) stmt_def gcost show ?thesis using stmt.psimps(5) g_def st'_def by simp
      qed
    qed
  qed
next
  case (6 i xe e cd st)
  define g where "g = costs (INVOKE i xe) e cd st"
  show ?case
  proof (rule allI[OF impI])
    fix st6' assume a1: "stmt (INVOKE i xe) e cd st = Normal ((), st6')"
    show "gas st6' ≤ gas st"
    proof (cases)
      assume "gas st ≤ g"
      with 6(1) a1 show ?thesis using stmt.psimps(6) g_def by simp
    next
      assume gcost: "¬ gas st ≤ g"
      then have l1: "assert Gas (λst. costs (INVOKE i xe) e cd st < gas st) st = Normal ((), st) "
using g_def by simp
      define st' where "st' = st(|gas := gas st - g|)"
      then have l2: "modify (λst. st(|gas := gas st - costs (INVOKE i xe) e cd st|)) st = Normal ((),
st')" using g_def by simp
      then show ?thesis
      proof (cases "ep $$ contract e")
        case None
        with 6(1) a1 gcost show ?thesis using stmt.psimps(6) g_def by simp
      next
        case (Some x)
        then have l3: "option Err (λ_. ep $$ contract e) st' = Normal (x, st')" by simp
        then show ?thesis
        proof (cases x)
          case (fields ct _ _)
          then show ?thesis
          proof (cases "fmlookup ct i")
            case None
            with 6(1) g_def a1 gcost Some fields show ?thesis using stmt.psimps(6) by simp
```

```
          next
            case s1: (Some a)
            then show ?thesis
            proof (cases a)
              case (Method x1)
              then show ?thesis
              proof (cases x1)
                case p1: (fields fp ext f)
                then show ?thesis
                proof (cases ext)
                  case True
                  with 6(1) a1 g_def gcost Some fields s1 Method p1 show ?thesis using stmt.psimps(6)
st'_def by auto
                next
                  case False
                  then have l4: "(case ct $$ i of None ⇒ throw Err | Some (Method (fp, True, f)) ⇒
throw Err
                       | Some (Method (fp, False, f)) ⇒ return (fp, f) | Some _ ⇒ throw Err) st' =
Normal ((fp,f),st')" using s1 Method p1 by simp
                  define m_o e'
                  where "m_o = memory st''"
                    and "e' = ffold (init ct) (emptyEnv (address e) (contract e) (sender e) (svalue e))
(fmdom ct)"
                  then show ?thesis
                  proof (cases "load False fp xe e' emptyStore emptyStore m_o e cd st' (gas st - g)")
                    case s4: (n a g')
                    define st'' where "st'' = st'⦇gas := g'⦈"
                    then show ?thesis
                    proof (cases a)
                      case f2: (fields e_l cd_l k_l m_l)
                      then have l5: "toState (load False fp xe e' emptyStore emptyStore m_o e cd) st' =
Normal ((e_l, cd_l, k_l, m_l), st'')" using st'_def st''_def s4 by simp
                      define k_o where "k_o = stack st'"
                      then show ?thesis
                      proof (cases "stmt f e_l cd_l (st''⦇stack:=k_l, memory:=m_l⦈)")
                        case n2: (n a st''')
                        with  a1 g_def gcost Some fields s1 Method p1 m_o_def e'_def s4 f2 k_o_def False
                        have "stmt (INVOKE i xe) e cd st = Normal ((), st'''⦇stack:=k_o⦈)"
                          using stmt.psimps(6)[OF 6(1)] st'_def st''_def by auto
                        with a1 have "gas st6' ≤ gas st'''" by auto
                        also from 6(2)[OF l1 l2 l3 fields l4 _ _ _ l5, where ?s'g="st''⦇stack := k_l,
memory := m_l⦈"] n2 m_o_def e'_def
                          have "... ≤ gas st'''" by auto
                        also from msel_ssel_expr_load_rexp_gas(4)[of False fp xe e' emptyStore
emptyStore m_o e cd st' "(gas st - g)"] have "... ≤ gas st'" using s4 st'_def st''_def f2 by auto
                        finally show ?thesis using st'_def by simp
                      next
                        case (e x)
                        with 6(1) a1 g_def gcost Some fields s1 Method p1 m_o_def e'_def s4 f2 show
?thesis using stmt.psimps(6) st'_def st''_def False by auto
                      qed
                    qed
                  next
                    case (e x)
                    with 6(1) a1 g_def gcost Some fields s1 Method p1 m_o_def e'_def show ?thesis using
stmt.psimps(6) st'_def False by auto
                  qed
                qed
              qed
            next
              case (Function _)
              with 6(1) g_def a1 gcost Some fields s1 show ?thesis using stmt.psimps(6) by simp
            next
              case (Var _)
```

```
            with 6(1) g_def a1 gcost Some fields s1 show ?thesis using stmt.psimps(6) by simp
          qed
        qed
      qed
    qed
  qed
next
  case (7 ad i xe val e cd st)
  define g where "g = costs (EXTERNAL ad i xe val) e cd st"
  show ?case
  proof (rule allI[OF impI])
    fix st' assume a1: "stmt (EXTERNAL ad i xe val) e cd st = Normal ((), st')"
    show "gas st' ≤ gas st"
    proof (cases)
      assume "gas st ≤ g"
      with 7(1) a1 show ?thesis using stmt.psimps(7) g_def by simp
    next
      assume gcost: "¬ gas st ≤ g"
      then have l1: "assert Gas (λst. costs (EXTERNAL ad i xe val) e cd st < gas st) st = Normal ((),
st) " using g_def by simp
      define st' where "st' = st⦇gas := gas st - g⦈"
      then have l2: " modify (λst. st⦇gas := gas st - costs (EXTERNAL ad i xe val) e cd st⦈) st =
Normal ((), st')" using g_def by simp
      then show ?thesis
      proof (cases "expr ad e cd st' (gas st - g)")
        case (n a0 g')
        define st'' where "st'' = st'⦇gas := g'⦈"
        with n have l3: "toState (expr ad e cd) st' = Normal (a0, st'')" using st'_def by simp
        then show ?thesis
        proof (cases a0)
          case (Pair b c)
          then show ?thesis
          proof (cases b)
            case (KValue adv)
            then show ?thesis
            proof (cases c)
              case (Value x1)
              then show ?thesis
              proof (cases x1)
                case (TSInt x1)
                with 7(1) g_def a1 gcost n Pair KValue Value show ?thesis using stmt.psimps(7) st'_def
by auto
              next
                case (TUInt x2)
                with 7(1) g_def a1 gcost n Pair KValue Value show ?thesis using stmt.psimps(7) st'_def
by auto
              next
                case TBool
                with 7(1) g_def a1 gcost n Pair KValue Value show ?thesis using stmt.psimps(7) st'_def
by auto
              next
                case TAddr
                then have l4: "(case a0 of (KValue adv, Value TAddr) ⇒ return adv | (KValue adv, Value
_) ⇒ throw Err | (KValue adv, _) ⇒ throw Err
                      | (_, b) ⇒ throw Err) st'' = Normal (adv, st'')" using Pair KValue Value by
simp
                then show ?thesis
                proof (cases "adv = address e")
                  case True
                  with 7(1) g_def a1 gcost n Pair KValue Value TAddr show ?thesis using stmt.psimps(7)
st'_def by auto
                next
                  case False
```

```
                then have l5: "assert Err (λ_. adv ≠ address e) st'' = Normal ((), st'')" by simp
                then show ?thesis
                proof (cases "type (accounts st'' adv)")
                  case None
                  with 7(1) g_def a1 gcost n Pair KValue Value TAddr False show ?thesis using
stmt.psimps(7) st'_def st''_def by auto
                next
                  case (Some x2)
                  then show ?thesis
                  proof (cases x2)
                    case EOA
                    with 7(1) g_def a1 gcost n Pair KValue Value TAddr False Some show ?thesis using
stmt.psimps(7) st'_def st''_def by auto
                  next
                    case (Contract c)
                    then have l6: "(λst. case type (accounts st adv) of Some (Contract c) ⇒ return c
st | _ ⇒ throw Err st) st'' = Normal (c, st'')" using Some by (simp split:atype.split option.split)
                    then show ?thesis
                    proof (cases "ep $$ c")
                      case None
                      with 7(1) g_def a1 gcost n Pair KValue Value TAddr False Contract Some show
?thesis using stmt.psimps(7) st'_def st''_def by auto
                    next
                      case s2: (Some x)
                      then show ?thesis
                      proof (cases x)
                        case p2: (fields ct x0 fb)
                        then have l7: "option Err (λ_. ep $$ c) st'' = Normal ((ct, x0, fb), st'')"
using s2 by simp
                        then show ?thesis
                        proof (cases "expr val e cd st'' (gas st'')")
                          case n1: (n kv g'')
                          define st''' where "st''' = st''⦇gas := g''⦈"
                          with n1 have l8: "toState (expr val e cd) st'' = Normal (kv, st''')" by
simp
                          then show ?thesis
                          proof (cases kv)
                            case p3: (Pair a b)
                            then show ?thesis
                            proof (cases a)
                              case k2: (KValue v)
                              then show ?thesis
                              proof (cases b)
                                case v: (Value t)
                                then have l9: "(case kv of (KValue v, Value t) ⇒ return (v, t) |
(KValue v, _) ⇒ throw Err | (_, b) ⇒ throw Err) st''' = Normal ((v,t), st''')" using n1 p3 k2 by simp
                                show ?thesis
                                proof (cases "convert t (TUInt 256) v")
                                  case None
                                  with 7(1) g_def a1 gcost n Pair KValue Value TAddr False Contract
Some s2 p2 None n1 p3 k2 v False show ?thesis using stmt.psimps(7)[OF 7(1)] st'_def st''_def st'''_def
by simp
                                next
                                  case s3: (Some v')
                                  define e' where "e' = ffold (init ct) (emptyEnv adv c (address e)
v') (fmdom ct)"
                                  show ?thesis
                                  proof (cases "fmlookup ct i")
                                    case None
                                    show ?thesis
                                    proof (cases "transfer (address e) adv v' (accounts st''')")
                                      case n2: None
                                      with 7(1) g_def a1 gcost n Pair KValue Value TAddr False
Contract Some s2 p2 None n1 p3 k2 v False s3 show ?thesis using stmt.psimps(7)[OF 7(1)] st'_def
```

```
st''_def st'''_def by simp
                                    next
                                      case s4: (Some acc)
                                      then have l10: "option Err (λst. transfer (address e) adv v'
(accounts st)) st''' = Normal (acc, st''')" by simp
                                      define k_o m_o
                                        where "k_o = stack st''''"
                                          and "m_o = memory st''''"
                                      show ?thesis
                                      proof (cases "stmt fb e' emptyStore (st'''⦇accounts := acc,
stack:=emptyStore, memory:=emptyStore⦈)")
                                        case n2: (n a st'''')
                                        with g_def a1 gcost n Pair KValue Value TAddr False Contract
Some s2 p2 None n1 p3 k2 v s4
                                        have "stmt (EXTERNAL ad i xe val) e cd st = Normal ((),
st''''⦇stack:=stack st''', memory := memory st'''⦈)"
                                          using stmt.psimps(7)[OF 7(1)] st'_def st''_def st'''_def
e'_def False s3 by simp
                                        with a1 have "gas st6' ≤ gas st''''" by auto
                                        also from 7(3)[OF l1 l2 l3 l4 l5 l6 l7 _ _ l8 l9 _ _ _ None
l10, where ?s'k="st''''" and ?s'l="st'''⦇accounts := acc,stack:=emptyStore, memory:=emptyStore⦈" and
?x=e', of "(x0,fb)" x0 fb] n2
                                          have "... ≤ gas (st'''⦇accounts := acc,stack:=emptyStore,
memory:=emptyStore⦈)" using e'_def s3 by simp
                                        also from msel_ssel_expr_load_rexp_gas(3)[of val e cd st''
"gas st'''"]
                                          have "... ≤ gas st''" using n1 st'_def st''_def st'''_def
p3 by simp
                                        also from msel_ssel_expr_load_rexp_gas(3)[of  ad e cd st'
"gas st - g"]
                                          have "... ≤ gas st'" using n st'_def st''_def st'''_def p3
by fastforce
                                        finally show ?thesis using st'_def by simp
                                      next
                                        case (e x)
                                        with 7(1) g_def a1 gcost n Pair KValue Value TAddr False Some
s2 Contract p2 None n1 p3 k2 v s4 s3 show ?thesis using stmt.psimps(7)[of ad i xe val e cd st] st'_def
st''_def st'''_def e'_def by simp
                                      qed
                                    qed
                                  next
                                    case s1: (Some a)
                                    then show ?thesis
                                    proof (cases a)
                                      case (Method x1)
                                      then show ?thesis
                                      proof (cases x1)
                                        case p4: (fields fp ext f)
                                        then show ?thesis
                                        proof (cases ext)
                                          case True
                                          then show ?thesis
                                          proof (cases "load True fp xe e' emptyStore emptyStore
emptyStore e cd st''' (gas st''')")
                                            case s4: (n a g''')
                                            define st'''' where "st'''' = st'''⦇gas := g'''⦈"
                                            then show ?thesis
                                            proof (cases a)
                                              case f1: (fields e_l cd_l k_l m_l)
                                              then have l10: "toState (load True fp xe e' emptyStore
emptyStore emptyStore e cd) st''' = Normal ((e_l, cd_l, k_l, m_l), st'''')" using s4 st''''_def by simp
                                              show ?thesis
                                              proof (cases "transfer (address e) adv v' (accounts
st'''')")
```

```
                                                    case n2: None
                                                    with 7(1) g_def a1 gcost n Pair KValue Value
TAddr False Some s2 Contract p2 s1 Method p4 n1 p3 k2 v s3 f1 e'_def True s4 show ?thesis using
stmt.psimps(7)[of ad i xe val e cd st] st'_def st''_def st'''_def st''''_def by simp
                                                  next
                                                    case s5: (Some acc)
                                                    then have l11: "option Err (λst. transfer (address e)
adv v' (accounts st)) st'''' = Normal (acc, st'''')" by simp
                                                    define k_o where "k_o = accounts st''''"
                                                    define m_o where "m_o = accounts st''''"
                                                    show ?thesis
                                                    proof (cases "stmt f e_l cd_l (st''''⦇accounts := acc,
stack:=k_l,memory:=m_l⦈)")
                                                      case n2: (n a st'''''')
                                                      with 7(1) g_def a1 gcost n Pair KValue Value TAddr
Some s2 Contract p2 s1 Method p4 n1 p3 k2 v k_o_def m_o_def e'_def s3 f1 s4 s5
                                                      have "stmt (EXTERNAL ad i xe val) e cd st = Normal
((), st''''⦇stack:=stack st'''', memory := memory st''''⦈)"
                                                        using stmt.psimps(7)[of ad i xe val e cd st]
st'_def st''_def st'''_def st''''_def True False by simp
                                                      with a1 have "gas st6' ≤ gas st''''''" by auto
                                                      also from 7(2)[OF l1 l2 l3 l4 l5 l6 l7 _ _ l8 l9 _
_ _ s1 Method _ _ _ l10 _ _ _ l11, where ?s'm="st''''⦇accounts := acc, stack := k_l, memory := m_l⦈" and
?s'l=st''''] p4 n2 e'_def True s3
                                                        have "... ≤ gas (st''''⦇accounts := acc, stack :=
k_l, memory := m_l⦈)" by simp
                                                      also from msel_ssel_expr_load_rexp_gas(4)[of True
fp xe e' emptyStore emptyStore emptyStore e cd st''' "gas st''''"]
                                                        have "... ≤ gas st'''" using s3 st'_def st''_def
st'''_def st''''_def f1 s4 by simp
                                                      also from msel_ssel_expr_load_rexp_gas(3)[of val e
cd st'' "gas st''"]
                                                        have "... ≤ gas st''" using n1 st'_def st''_def
st'''_def by fastforce
                                                      also from msel_ssel_expr_load_rexp_gas(3)[of ad e
cd st' "gas st - g"]
                                                        have "... ≤ gas st'" using n st'_def st''_def
st'''_def by fastforce
                                                      finally show ?thesis using st'_def by simp
                                                    next
                                                      case (e x)
                                                      with 7(1) g_def a1 gcost n Pair KValue Value TAddr
False Some s2 Contract p2 s1 Method p4 n1 p3 k2 v k_o_def m_o_def e'_def s3 f1 s4 s5 True show ?thesis
using stmt.psimps(7) st'_def st''_def st'''_def st''''_def by simp
                                                    qed
                                                  qed
                                                qed
                                              next
                                                case (e x)
                                                with 7(1) g_def a1 gcost n Pair KValue Value TAddr False
Some s2 Contract p2 s1 Method p4 n1 p3 k2 v e'_def True s3 show ?thesis using stmt.psimps(7) st'_def
st''_def st'''_def by simp
                                              qed
                                            next
                                              case f: False
                                              with 7(1) g_def a1 gcost n Pair KValue Value TAddr False
Some s2 Contract p2 s1 Method p4 n1 p3 k2 v s3 show ?thesis using stmt.psimps(7) st'_def st''_def
st'''_def by simp
                                            qed
                                          qed
                                        next
                                          case (Function _)
                                          with 7(1) g_def a1 gcost n Pair KValue Value TAddr False Some
s2 Contract p2 s1 n1 p3 k2 v s3 show ?thesis using stmt.psimps(7) st'_def st''_def st'''_def by simp
```

```
                                    next
                                      case (Var _)
                                      with 7(1) g_def a1 gcost n Pair KValue Value TAddr False Some
s2 Contract p2 s1 n1 p3 k2 v s3 show ?thesis using stmt.psimps(7) st'_def st''_def st'''_def by simp
                                    qed
                                  qed
                                qed
                              next
                                case (Calldata x2)
                                with 7(1) g_def a1 gcost n Pair KValue Value TAddr False Some s2
Contract p2 n1 p3 k2 show ?thesis using stmt.psimps(7) st'_def st''_def st'''_def by simp
                              next
                                case (Memory x3)
                                with 7(1) g_def a1 gcost n Pair KValue Value TAddr False Some s2
Contract p2 n1 p3 k2 show ?thesis using stmt.psimps(7) st'_def st''_def st'''_def by simp
                              next
                                case (Storage x4)
                                with 7(1) g_def a1 gcost n Pair KValue Value TAddr False Some s2
Contract p2 n1 p3 k2 show ?thesis using stmt.psimps(7) st'_def st''_def st'''_def by simp
                              qed
                            next
                              case (KCDptr x2)
                              with 7(1) g_def a1 gcost n Pair KValue Value TAddr False Some s2
Contract p2 n1 p3 show ?thesis using stmt.psimps(7) st'_def st''_def st'''_def by simp
                            next
                              case (KMemptr x3)
                              with 7(1) g_def a1 gcost n Pair KValue Value TAddr False Some s2
Contract p2 n1 p3 show ?thesis using stmt.psimps(7) st'_def st''_def st'''_def by simp
                            next
                              case (KStoptr x4)
                              with 7(1) g_def a1 gcost n Pair KValue Value TAddr False Some s2
Contract p2 n1 p3 show ?thesis using stmt.psimps(7) st'_def st''_def st'''_def by simp
                            qed
                          qed
                        next
                          case n2: (e x)
                          with 7(1) g_def a1 gcost n Pair KValue Value TAddr False Some s2 Contract
p2 show ?thesis using stmt.psimps(7) st'_def st''_def by simp
                        qed
                      qed
                    qed
                  qed
                qed
              qed
            qed
          next
            case (Calldata x2)
            with 7(1) g_def a1 gcost n Pair KValue show ?thesis using stmt.psimps(7) st'_def
st''_def by simp
          next
            case (Memory x3)
            with 7(1) g_def a1 gcost n Pair KValue show ?thesis using stmt.psimps(7) st'_def
st''_def by simp
          next
            case (Storage x4)
            with 7(1) g_def a1 gcost n Pair KValue show ?thesis using stmt.psimps(7) st'_def
st''_def by simp
          qed
        next
          case (KCDptr x2)
          with 7(1) g_def a1 gcost n Pair show ?thesis using stmt.psimps(7) st'_def st''_def by
simp
        next
          case (KMemptr x3)
```

```
          with 7(1) g_def a1 gcost n Pair show ?thesis using stmt.psimps(7) st'_def st''_def by
simp
        next
          case (KStoptr x4)
          with 7(1) g_def a1 gcost n Pair show ?thesis using stmt.psimps(7) st'_def st''_def by
simp
        qed
      qed
    next
      case (e _)
      with 7(1) g_def a1 gcost show ?thesis using stmt.psimps(7) st'_def by simp
    qed
  qed
qed
next
  case (8 ad ex e cd st)
  define g where "g = costs (TRANSFER ad ex) e cd st"
  show ?case
  proof (rule allI[OF impI])
    fix st6' assume stmt_def: "stmt (TRANSFER ad ex) e cd st = Normal ((), st6')"
    show "gas st6' ≤ gas st"
    proof cases
      assume "gas st ≤ g"
      with 8 stmt_def g_def show ?thesis using stmt.psimps(8)[of ad ex e cd st] by simp
    next
      assume "¬ gas st ≤ g"
      then have l1: "assert Gas (λst. costs (TRANSFER ad ex) e cd st < gas st) st = Normal ((), st) "
using g_def by simp
      define st' where "st' = st⦇gas := gas st - g⦈"
      then have l2: " modify (λst. st⦇gas := gas st - costs (TRANSFER ad ex) e cd st⦈) st = Normal
((), st')" using g_def by simp
      show ?thesis
      proof (cases "expr ad e cd st' (gas st - g)")
        case (n a0 g')
        define st'' where "st'' = st'⦇gas := g'⦈"
        with n have l3: "toState (expr ad e cd) st' = Normal (a0, st'')" using st'_def by simp
        then show ?thesis
        proof (cases a0)
          case (Pair b c)
          then show ?thesis
          proof (cases b)
            case (KValue adv)
            then show ?thesis
            proof (cases c)
              case (Value x1)
              then show ?thesis
              proof (cases x1)
                case (TSInt x1)
                with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value g_def show ?thesis using
stmt.psimps(8) st'_def st''_def by simp
              next
                case (TUInt x2)
                with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value g_def show ?thesis using
stmt.psimps(8) st'_def st''_def by simp
              next
                case TBool
                with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value g_def show ?thesis using
stmt.psimps(8) st'_def st''_def by simp
              next
                case TAddr
                then have l4: "(case a0 of (KValue adv, Value TAddr) ⇒ return adv | (KValue adv, Value
_) ⇒ throw Err | (KValue adv, _) ⇒ throw Err
                      | (_, b) ⇒ throw Err) st'' = Normal (adv, st'')" using Pair KValue Value by
simp
```

```
                then show ?thesis
                proof (cases "expr ex e cd st'' (gas st'')")
                  case n2: (n a1 g'')
                  define st''' where "st''' = st''⦇gas := g''⦈"
                  with n2 have l5: "toState (expr ex e cd) st'' = Normal (a1, st''')" by simp
                  then show ?thesis
                  proof (cases a1)
                    case p2: (Pair b c)
                    then show ?thesis
                    proof (cases b)
                      case k2: (KValue v)
                      then show ?thesis
                      proof (cases c)
                        case v2: (Value t)
                        then have l6: "(case a1 of (KValue v, Value t) ⇒ return (v, t) | (KValue v, _)
⇒ throw Err | (_, b) ⇒ throw Err) st''' = Normal ((v,t), st''')" using p2 k2 by simp
                        then show ?thesis
                        proof (cases "convert t (TUInt 256) v")
                          case None
                          with 8(1) stmt_def g_def ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p2 k2 v2
TAddr show ?thesis using stmt.psimps(8) st'_def st''_def st'''_def by simp
                        next
                          case (Some v')
                          then show ?thesis
                          proof (cases "type (accounts st''' adv)")
                            case None
                            with 8(1) stmt_def g_def ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p2 k2 v2
TAddr Some show ?thesis using stmt.psimps(8) st'_def st''_def st'''_def by simp
                          next
                            case s0: (Some a)
                            then show ?thesis
                            proof (cases a)
                              case EOA
                              then show ?thesis
                              proof (cases "transfer (address e) adv v' (accounts st''')")
                                case None
                                with 8(1) stmt_def g_def ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p2 k2
v2 TAddr Some EOA s0 show ?thesis using stmt.psimps(8) st'_def st''_def st'''_def by simp
                              next
                                case s1: (Some acc)
                                then have l7: "option Err (λst. transfer (address e) adv v' (accounts
st)) st''' = Normal (acc, st''')" using st'''_def by simp
                                with 8(1) ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p2 k2 v2 TAddr Some
EOA g_def s0
                                  have "stmt (TRANSFER ad ex) e cd st = Normal
((),st'''⦇accounts:=acc⦈)" using stmt.psimps(8)[of ad ex e cd st] st'_def st''_def st'''_def by simp
                                with stmt_def have "gas st6' = gas (st'''⦇accounts:=acc⦈)" by auto
                                also from msel_ssel_expr_load_rexp_gas(3)[of ex e cd st'' "gas st''"]
                                  have "... ≤ gas st''" using st'_def st''_def st'''_def n2 by
fastforce
                                also from msel_ssel_expr_load_rexp_gas(3)[of ad e cd st' "gas st - g"]
                                  have "... ≤ gas st'" using st'_def st''_def n by fastforce
                                  finally show ?thesis using st'_def by simp
                              qed
                            next
                              case (Contract c)
                              then show ?thesis
                              proof (cases "ep $$ c")
                                case None
                                with 8(1) stmt_def g_def ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p2 k2
v2 TAddr Contract Some s0 show ?thesis using stmt.psimps(8) st'_def st''_def st'''_def by simp
                              next
                                case s2: (Some a)
                                then show ?thesis
```

```
                                proof (cases a)
                                  case p3: (fields ct cn f)
                                  with s2 have l7: "option Err (λ_. ep $$ c) st''' = Normal ((ct, cn,
f), st''')" by simp
                                  define e' where "e' = ffold_init ct (emptyEnv adv c (address e) v')
(fmdom ct)"
                                  show ?thesis
                                  proof (cases "transfer (address e) adv v' (accounts st''')")
                                    case None
                                    with 8(1) stmt_def g_def ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p2
k2 v2 TAddr Contract Some s2 p3 s0 show ?thesis using stmt.psimps(8) st'_def st''_def st'''_def by
simp
                                  next
                                    case s3: (Some acc)
                                    then have l8: "option Err (λst. transfer (address e) adv v'
(accounts st)) st''' = Normal (acc, st''')" by simp
                                    then show ?thesis
                                    proof (cases "stmt f e' emptyStore (st'''⦇accounts := acc,
stack:=emptyStore, memory:=emptyStore⦈)")
                                      case n3: (n a st'''')
                                      with 8(1) ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p2 k2 v2 TAddr
Some s2 p3 g_def Contract s3 s0
                                      have "stmt (TRANSFER ad ex) e cd st = Normal
((),st''''⦇stack:=stack st''', memory := memory st'''⦈)" using e'_def stmt.psimps(8)[of ad ex e cd
st] st'_def st''_def st'''_def by simp
                                      with stmt_def have "gas st6' ≤ gas st''''" by auto
                                      also from 8(2)[OF l1 l2 l3 l4 l5 l6, of v t _ _ "accounts st'''"
"st'''", OF _ _ _ s0 Contract l7 _ _ _ _ _ l8, where ?s'k="st'''⦇accounts := acc, stack := emptyStore,
memory := emptyStore⦈"] ‘¬ gas st ≤ g‘ e'_def n3 Some
                                        have "... ≤ gas (st'''⦇accounts := acc, stack := emptyStore,
memory := emptyStore⦈)" by simp
                                      also from msel_ssel_expr_load_rexp_gas(3)[of ex e cd st'' "gas
st'''"]
                                        have "... ≤ gas st'''" using st'_def st''_def st'''_def n2 by
fastforce
                                      also from msel_ssel_expr_load_rexp_gas(3)[of ad e cd st' "gas st
- g"]
                                        have "... ≤ gas st'" using st'_def st''_def n by fastforce
                                      finally show ?thesis using st'_def by simp
                                    next
                                      case (e x)
                                      with 8(1) ‘¬ gas st ≤ g‘ n Pair KValue Value n2 p2 k2 v2 TAddr
Some s2 p3 g_def e'_def stmt_def Contract s3 s0 show ?thesis using stmt.psimps(8)[of ad ex e cd st]
st'_def st''_def st'''_def by simp
                                    qed
                                  qed
                                qed
                              qed
                            qed
                          qed
                        qed
                      next
                        case (Calldata x2)
                        with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value TAddr n2 p2 k2 g_def
show ?thesis using stmt.psimps(8) st'_def st''_def st'''_def by simp
                      next
                        case (Memory x3)
                        with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value TAddr n2 p2 k2 g_def
show ?thesis using stmt.psimps(8) st'_def st''_def st'''_def by simp
                      next
                        case (Storage x4)
                        with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value TAddr n2 p2 k2 g_def
show ?thesis using stmt.psimps(8) st'_def st''_def st'''_def by simp
                      qed
```

```
                    next
                      case (KCDptr x2)
                      with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value TAddr n2 p2 g_def show
?thesis using stmt.psimps(8) st'_def st''_def st'''_def by simp
                    next
                      case (KMemptr x3)
                      with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value TAddr n2 p2 g_def show
?thesis using stmt.psimps(8) st'_def st''_def st'''_def by simp
                    next
                      case (KStoptr x4)
                      with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value TAddr n2 p2 g_def show
?thesis using stmt.psimps(8) st'_def st''_def st'''_def by simp
                    qed
                  qed
                next
                  case (e e)
                  with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue Value TAddr g_def show ?thesis
using stmt.psimps(8) st'_def st''_def by simp
                qed
              qed
            next
              case (Calldata x2)
              with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue g_def show ?thesis using stmt.psimps(8)
st'_def st''_def by simp
            next
              case (Memory x3)
              with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue g_def show ?thesis using stmt.psimps(8)
st'_def st''_def by simp
            next
              case (Storage x4)
              with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair KValue g_def show ?thesis using stmt.psimps(8)
st'_def st''_def by simp
            qed
          next
            case (KCDptr x2)
            with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair g_def show ?thesis using stmt.psimps(8) st'_def
st''_def by simp
          next
            case (KMemptr x3)
            with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair g_def show ?thesis using stmt.psimps(8) st'_def
st''_def by simp
          next
            case (KStoptr x4)
            with 8(1) stmt_def ‘¬ gas st ≤ g‘ n Pair g_def show ?thesis using stmt.psimps(8) st'_def
st''_def by simp
          qed
        qed
      next
        case (e e)
        with 8(1) stmt_def ‘¬ gas st ≤ g‘ g_def show ?thesis using stmt.psimps(8) st'_def by simp
      qed
    qed
  qed
next
  case (9 id0 tp s e_v cd st)
  define g where "g = costs (BLOCK ((id0, tp), None) s) e_v cd st"
  show ?case
  proof (rule allI[OF impI])
    fix st6' assume stmt_def: "stmt (BLOCK ((id0, tp), None) s) e_v cd st = Normal ((), st6')"
    show "gas st6' ≤ gas st"
    proof cases
      assume "gas st ≤ g"
      with 9 stmt_def g_def show ?thesis using stmt.psimps(9) by simp
    next
```

```
      assume "¬ gas st ≤ g"
      then have l1: "assert Gas (λst. costs (BLOCK ((id0, tp), None) s) e_v cd st < gas st) st = Normal
((), st) " using g_def by simp
      define st' where "st' = st(|gas := gas st - g|)"
      then have l2: "modify (λst. st(|gas := gas st - costs (BLOCK ((id0, tp), None) s) e_v cd st|)) st =
Normal ((), st')" using g_def by simp
      show ?thesis
      proof (cases "decl id0 tp None False cd (memory st') (storage st') (cd, (memory st'), (stack
st'), e_v)")
        case n2: None
        with 9 stmt_def ‘¬ gas st ≤ g‘ g_def show ?thesis using stmt.psimps(9) st'_def by simp
      next
        case (Some a)
        then have l3: "option Err (λst. decl id0 tp None False cd (memory st) (storage st) (cd, memory
st, stack st, e_v)) st' = Normal (a, st')" by simp
        then show ?thesis
        proof (cases a)
          case (fields cd' mem' sck' e')
          with 9(1) stmt_def ‘¬ gas st ≤ g‘ g_def have "stmt (BLOCK ((id0, tp), None) s) e_v cd st
= stmt s e' cd' (st(|gas := gas st - g, stack := sck', memory := mem'|))" using stmt.psimps(9)[OF 9(1)]
Some st'_def by simp
          with 9(2)[OF l1 l2 l3] stmt_def ‘¬ gas st ≤ g‘ fields g_def have "gas st6' ≤ gas (st(|gas :=
gas st - g, stack := sck', memory := mem'|))" using st'_def by fastforce
          then show ?thesis by simp
        qed
      qed
    qed
  qed
next
  case (10 id0 tp ex' s e_v cd st)
  define g where "g = costs (BLOCK ((id0, tp), Some ex') s) e_v cd st"
  show ?case
  proof (rule allI[OF impI])
    fix st6' assume stmt_def: "stmt (BLOCK ((id0, tp), Some ex') s) e_v cd st = Normal ((), st6')"
    show "gas st6' ≤ gas st"
    proof cases
      assume "gas st ≤ g"
      with 10 stmt_def g_def show ?thesis using stmt.psimps(10) by simp
    next
      assume "¬ gas st ≤ g"
      then have l1: "assert Gas (λst. costs (BLOCK ((id0, tp), Some ex') s) e_v cd st < gas st) st =
Normal ((), st) " using g_def by simp
      define st' where "st' = st(|gas := gas st - g|)"
      then have l2: "modify (λst. st(|gas := gas st - costs (BLOCK ((id0, tp), Some ex') s) e_v cd st|))
st = Normal ((), st')" using g_def by simp
      show ?thesis
      proof (cases "expr ex' e_v cd st' (gas st - g)")
        case (n a g')
        define st'' where "st'' = st'(|gas := g'|)"
        with n have l3: "toState (expr ex' e_v cd) st' = Normal (a, st'')" using st''_def st'_def by
simp
        then show ?thesis
        proof (cases a)
          case (Pair v t)
          then show ?thesis
          proof (cases "decl id0 tp (Some (v, t)) False cd (memory st'') (storage st'') (cd, memory
st'', stack st'', e_v)")
            case None
            with 10(1) stmt_def ‘¬ gas st ≤ g‘ n Pair g_def show ?thesis using stmt.psimps(10)
st'_def st''_def by simp
          next
            case s2: (Some a)
            then have l4: "option Err (λst. decl id0 tp (Some (v, t)) False cd (memory st) (storage st)
(cd, memory st, stack st, e_v)) st'' = Normal (a, st'')" by simp
```

```
            then show ?thesis
            proof (cases a)
              case (fields cd' mem' sck' e')
              with 10(1) stmt_def ‘¬ gas st ≤ g‘ n Pair s2 g_def have "stmt (BLOCK ((id0, tp), Some
ex') s) e_v cd st = stmt s e' cd' (st''⦇stack := sck', memory := mem'⦈)" using stmt.psimps(10)[of id0 tp
ex' s e_v cd st] st'_def st''_def by simp
              with 10(2)[OF l1 l2 l3 Pair l4 fields, where s'd="st''⦇stack := sck', memory := mem'⦈"]
n stmt_def ‘¬ gas st ≤ g‘ s2 fields g_def have "gas st6' ≤ gas st''" by simp
              moreover from msel_ssel_expr_load_rexp_gas(3)[of ex' e_v cd st' "gas st - g"] n have
"gas st'' ≤ gas st'" using st'_def st''_def by fastforce
              ultimately show ?thesis using st'_def by simp
            qed
          qed
        qed
      next
        case (e e)
        with 10 stmt_def ‘¬ gas st ≤ g‘ g_def show ?thesis using stmt.psimps(10) st'_def by simp
      qed
    qed
  qed
next
  case (11 i xe val e cd st)
  define g where "g = costs (NEW i xe val) e cd st"
  show ?case
  proof (rule allI[OF impI])
    fix st6' assume a1: "stmt (NEW i xe val) e cd st = Normal ((), st6')"
    show "gas st6' ≤ gas st"
    proof (cases)
      assume "gas st ≤ g"
      with 11(1) a1 show ?thesis using stmt.psimps(11) g_def by simp
    next
      assume gcost: "¬ gas st ≤ g"
      then have l1: "assert Gas (λst. costs (NEW i xe val) e cd st < gas st) st = Normal ((), st) "
using g_def by simp
      define st' where "st' = st⦇gas := gas st - g⦈"
      then have l2: "modify (λst. st⦇gas := gas st - costs (NEW i xe val) e cd st⦈) st = Normal ((),
st')" using g_def by simp
      define adv where "adv = hash (address e) (ShowL_nat (contracts (accounts st' (address e))))"
      then show ?thesis
      proof (cases "type (accounts st' adv) = None")
        case True
        then show ?thesis
        proof (cases "expr val e cd st' (gas st')")
          case n0: (n kv g')
          define st'' where "st'' = st'⦇gas := g'⦈"
          then have l4: "toState (expr val e cd) st' = Normal (kv, st'')" using n0 by simp
          then show ?thesis
          proof (cases kv)
            case p0: (Pair a b)
            then show ?thesis
            proof (cases a)
              case k0: (KValue v)
              then show ?thesis
              proof (cases b)
                case v0: (Value t)
                then show ?thesis
                proof (cases "ep $$ i")
                  case None
                  with a1 gcost g_def True n0 p0 k0 v0
                  show ?thesis using stmt.psimps(11)[OF 11(1)] adv_def st'_def st''_def by simp
                next
                  case s0: (Some a)
                  then have l5: "option Err (λ_. ep $$ i) st'' = Normal (a, st'')" by simp
                  then show ?thesis
```

```
                proof (cases a)
                  case f0: (fields ct cn _)
                  define e' where "e' = ffold_init ct (emptyEnv adv i (address e) v) (fmdom ct)"
                  then show ?thesis
                  proof (cases "load True (fst cn) xe e' emptyStore emptyStore emptyStore e cd st''
(gas st'')")
                    case n1: (n a g'')
                    define st''' where "st''' = st''⦇gas := g''⦈"
                    then have l6: "toState (load True (fst cn) xe e' emptyStore emptyStore emptyStore
e cd) st'' = Normal (a, st''')" using n1 by simp
                    then show ?thesis
                    proof (cases a)
                      case f1: (fields e_l cd_l k_l m_l)
                      define st'''' where "st'''' = st'''⦇accounts:=(accounts st''')(adv := ⦇bal =
ShowL_int 0, type = Some (Contract i), contracts = 0⦈), storage:=(storage st''')(adv := {$$})⦈"
                      then show ?thesis
                      proof (cases "transfer (address e) adv v (accounts st'''')")
                        case None
                        with a1 gcost g_def True n0 p0 k0 v0 s0 f0 n1 f1
                        show ?thesis using stmt.psimps(11)[OF 11(1)] adv_def e'_def st'_def st''_def
st'''_def st''''_def by (simp add:Let_def)
                      next
                        case s1: (Some acc)
                        define st''''' where "st''''' = st''''⦇accounts := acc, stack:=k_l,
memory:=m_l⦈"
                        then show ?thesis
                        proof (cases "stmt (snd cn) e_l cd_l st'''''")
                          case (n a st'''''')
                          define st''''''' where "st''''''' = st''''''⦇stack:=stack st'''', memory :=
memory st''''⦈"
                          define st'''''''' where "st'''''''' = incrementAccountContracts (address e)
st'''''''"
                          from a1 gcost g_def True n0 p0 k0 v0 s0 f0 n1 f1 s1 n have "st6' =
st''''''''"
                            using st'_def st''_def st'''_def st''''_def st'''''_def st'''''''_def
st''''''''_def
                            stmt.psimps(11)[OF 11(1)] adv_def e'_def by (simp add:Let_def)
                          then have "gas st6' = gas st''''''''" by simp
                          also have "... ≤ gas st'''''''" using st''''''''_def
incrementAccountContracts_def by simp
                          also have "... ≤ gas st''''''" using st'''''''_def by simp
                          also have "... ≤ gas st'''''" using 11(2)[OF l1 l2 _ _ l4 _ _ l5 _ _ e'_def
l6, where ?s'h="st''''" and ?s'i="st''''" and ?s'j="st''''" and ?s'k="st''''⦇accounts := acc, stack
:= k_l, memory := m_l⦈", of st' "()"] p0 k0 v0 f0 f1 s1 n True st''''_def st'''''_def adv_def by simp
                          also have "... ≤ gas st''''" using st'''''_def by simp
                          also have "... ≤ gas st'''" using st''''_def by simp
                          also have "... ≤ gas st''" using st'''_def msel_ssel_expr_load_rexp_gas(4)
n1 f1 by simp
                          also have "... ≤ gas st'" using st''_def msel_ssel_expr_load_rexp_gas(3) n0
p0 by simp
                          also have "... ≤ gas st" using st'_def by simp
                          finally show ?thesis .
                        next
                          case (e e)
                          with a1 gcost g_def n0 True p0 k0 v0 s0 f0 n1 f1 s1
                          show ?thesis using stmt.psimps(11)[OF 11(1)] adv_def e'_def st'_def
st''_def st'''_def st''''_def st'''''_def by (simp add:Let_def)
                        qed
                      qed
                    qed
                  next
                    case (e e)
                    with a1 gcost g_def n0 True p0 k0 v0 s0 f0
                    show ?thesis using stmt.psimps(11)[OF 11(1)] adv_def e'_def st'_def st''_def by
```

```
(simp add:Let_def)
                    qed
                  qed
                qed
              next
                case (Calldata x2)
                with a1 gcost g_def n0 True p0 k0
                show ?thesis using stmt.psimps(11)[OF 11(1)] adv_def st'_def by simp
              next
                case (Memory x3)
                with a1 gcost g_def n0 True p0 k0
                show ?thesis using stmt.psimps(11)[OF 11(1)] adv_def st'_def by simp
              next
                case (Storage x4)
                with a1 gcost g_def n0 True p0 k0
                show ?thesis using stmt.psimps(11)[OF 11(1)] adv_def st'_def by simp
              qed
            next
              case (KCDptr x2)
              with a1 gcost g_def n0 True p0
              show ?thesis using stmt.psimps(11)[OF 11(1)] adv_def st'_def by simp
            next
              case (KMemptr x3)
              with a1 gcost g_def n0 True p0
              show ?thesis using stmt.psimps(11)[OF 11(1)] adv_def st'_def by simp
            next
              case (KStoptr x4)
              with a1 gcost g_def n0 True p0
              show ?thesis using stmt.psimps(11)[OF 11(1)] adv_def st'_def by simp
            qed
          qed
        next
          case (e e)
          with a1 gcost g_def True
          show ?thesis using stmt.psimps(11)[OF 11(1)] adv_def st'_def by simp
        qed
      next
        case False
        with a1 gcost g_def
        show ?thesis using stmt.psimps(11)[OF 11(1)] adv_def st'_def by (simp split:if_split_asm)
      qed
    qed
  qed
qed
```

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
  apply (relation "measures [sgas, ssize]")
  apply (auto split: if_split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm
option.split_asm Member.split_asm bool.split_asm atype.split_asm)
  apply ((stmt_dom_gas | msel_ssel_expr_load_rexp)+, costs?, simp)+
  done
```

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)"
  apply (induct rule: stmt.induct[where ?P="λs6 ev6 cd6 st6. stmt_dom (s6, ev6, cd6, st6)"])
  apply (simp_all add: stmt.domintros(1-10))
  apply (rule stmt.domintros(11), force)
  done

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|)"
  using assms by (auto split:if_split_asm)

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 -
  from assms consider
    (1) v 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 ((KValue v, Value t), g)"
  | (2) p x 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 ((KCDptr p, Calldata (MTArray x t)), g)"
  | (3) p x 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)"
  | (4) p x 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)"
  | (5) p t 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)"
    by (auto split:if_split_asm result.split_asm Stackvalue.split_asm Type.split_asm MTypes.split_asm
STypes.split_asm)
  then show ?thesis
  proof cases
    case 1
    with assms consider
      (11) l t' g' where "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LStackloc l, Value t'),g')"
    | (12) l t' g' where "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LStoreloc l, Storage (STValue
t')),g')"
    | (13) l t' g' where "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LMemloc l, Memory (MTValue
t')),g')"
      by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm)
    then show ?thesis
    proof cases
      case 11
      with 1 assms show ?thesis using that(1) by (auto split:if_split_asm result.split_asm
Type.split_asm LType.split_asm MTypes.split_asm STypes.split_asm option.split_asm)
    next
      case 12
      with 1 assms show ?thesis using that(2) by (auto split:if_split_asm result.split_asm
Type.split_asm LType.split_asm MTypes.split_asm STypes.split_asm option.split_asm)
    next
      case 13
      with 1 assms show ?thesis using that(3) by (auto split:if_split_asm result.split_asm
Type.split_asm LType.split_asm MTypes.split_asm STypes.split_asm option.split_asm)
    qed
  next
    case 2
    with assms consider
      (21) l t' g' where "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LStackloc l, Memory t'),g')"
    | (22) l t' g' where "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LStackloc l, Storage t'),g')"
    | (23) l t' g' where "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LStoreloc l, t'),g')"
    | (24) l t' g' where "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LMemloc l, t'),g')"
      by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm)
    then show ?thesis
    proof cases
      case 21
      moreover from assms 2 21 obtain p' where 3: "accessStore l (stack st) = Some (KMemptr p')"
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm option.split_asm Stackvalue.split_asm)
      moreover from assms 2 21 3 obtain m where "cpm2m p p' x t cd (memory st) = Some m"
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
```

```
STypes.split_asm option.split_asm Stackvalue.split_asm)
      ultimately show ?thesis using that(4) assms 2 21
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm option.split_asm Stackvalue.split_asm)
    next
      case 22
      moreover from assms 2 22 obtain p' where 3: "accessStore l (stack st) = Some (KStoptr p')"
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm option.split_asm Stackvalue.split_asm)
      moreover from assms 2 22 3 4 obtain s where "cpm2s p p' x t cd (storage st (address ev)) =
Some s"
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm option.split_asm Stackvalue.split_asm)
      ultimately show ?thesis using that(5) assms 2 22
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm option.split_asm Stackvalue.split_asm)
    next
      case 23
      moreover from assms 2 23 3 4 obtain s where "cpm2s p l x t cd (storage st (address ev)) = Some
s"
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm option.split_asm Stackvalue.split_asm)
      ultimately show ?thesis using that(6) assms 2
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm option.split_asm Stackvalue.split_asm)
    next
      case 24
      moreover from assms 2 24 obtain m where "cpm2m p l x t cd (memory st) = Some m"
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm option.split_asm Stackvalue.split_asm)
      ultimately show ?thesis using that(7) assms 2
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm option.split_asm Stackvalue.split_asm)
    qed
  next
    case 3
    with assms consider
      (31) l t' g' where "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LStackloc l, Memory t'),g')"
    | (32) l t' g' where "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LStackloc l, Storage t'),g')"
    | (33) l t' g' where "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LStoreloc l, t'),g')"
    | (34) l t' g' where "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LMemloc l, t'),g')"
      by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm)
    then show ?thesis
    proof cases
      case 31
      then show ?thesis using that(8) assms 3 by (auto split:if_split_asm)
    next
      case 32
      moreover from assms 3 32 obtain p' where 4: "accessStore l (stack st) = Some (KStoptr p')"
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm option.split_asm Stackvalue.split_asm)
      moreover from assms 3 32 4 5 obtain s where "cpm2s p p' x t (memory st) (storage st (address
ev)) = Some s"
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm option.split_asm Stackvalue.split_asm)
      ultimately show ?thesis using that(9) assms 3
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
STypes.split_asm option.split_asm Stackvalue.split_asm)
    next
      case 33
      moreover from assms 3 33 3 4 obtain s where "cpm2s p l x t (memory st) (storage st (address
ev)) = Some s"
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
```

```
SType.split_asm option.split_asm Stackvalue.split_asm)
      ultimately show ?thesis using that(10) assms 3
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
SType.split_asm option.split_asm Stackvalue.split_asm)
    next
      case 34
      then show ?thesis using that(11) assms 3
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
SType.split_asm option.split_asm Stackvalue.split_asm)
    qed
  next
    case 4
    with assms consider
      (41) l t' g' where "lexp lv ev cd (st(|gas := g|)) g = Normal((LStackloc l, Memory t'),g')"
    | (42) l t' g' where "lexp lv ev cd (st(|gas := g|)) g = Normal((LStackloc l, Storage t'),g')"
    | (43) l t' g' where "lexp lv ev cd (st(|gas := g|)) g = Normal((LStoreloc l, t'),g')"
    | (44) l t' g' where "lexp lv ev cd (st(|gas := g|)) g = Normal((LMemloc l, t'),g')"
      by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
SType.split_asm)
    then show ?thesis
    proof cases
      case 41
      moreover from assms 4 41 obtain p' where 5: "accessStore l (stack st) = Some (KMemptr p')"
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
SType.split_asm option.split_asm Stackvalue.split_asm)
      moreover from assms 4 41 5 6 obtain m where "cps2m p p' x t (storage st (address ev)) (memory
st) = Some m"
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
SType.split_asm option.split_asm Stackvalue.split_asm)
      ultimately show ?thesis using that(12) assms 4
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
SType.split_asm option.split_asm Stackvalue.split_asm)
    next
      case 42
      then show ?thesis using that(13) assms 4
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
SType.split_asm option.split_asm Stackvalue.split_asm)
    next
      case 43
      moreover from assms 4 43 5 obtain s where "copy p l x t (storage st (address ev)) = Some s"
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
SType.split_asm option.split_asm Stackvalue.split_asm)
      ultimately show ?thesis using that(14) assms 4
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
SType.split_asm option.split_asm Stackvalue.split_asm)
    next
      case 44
      moreover from assms 4 44 5 obtain m where "cps2m p l x t (storage st (address ev)) (memory st)
= Some m"
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
SType.split_asm option.split_asm Stackvalue.split_asm)
      ultimately show ?thesis using that(15) assms 4
        by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
SType.split_asm option.split_asm Stackvalue.split_asm)
    qed
  next
    case 5
    then show ?thesis using that(16) assms
      by (auto split:if_split_asm result.split_asm Type.split_asm LType.split_asm MTypes.split_asm
SType.split_asm option.split_asm Stackvalue.split_asm)
  qed
qed

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')"
  using assms by (simp split:if_split_asm result.split_asm prod.split_asm)

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')"
  using assms by (simp split:if_split_asm result.split_asm prod.split_asm Stackvalue.split_asm
Type.split_asm Types.split_asm)

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⦈"
  using assms
proof -
  from assms have 1: "gas st > costs (WHILE ex s0) ev cd st" by (simp split:if_split_asm)
  moreover from assms 1 have 2: "modify (λst. st⦇gas := gas st - costs (WHILE ex s0) ev cd st⦈) st =
Normal ((), st⦇gas := gas st - costs (WHILE ex s0) ev cd st⦈)" by simp
  moreover from assms 1 2 obtain b g where 3: "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 b, Value TBool), g)" by (simp
split:result.split_asm prod.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm)
  ultimately consider (True) "b = ShowL_bool True" | (False) "b = ShowL_bool False" | (None) "b ≠
ShowL_bool True ∧ b ≠ ShowL_bool False" by auto
  then show ?thesis
  proof cases
    case True
    moreover from assms 1 2 3 True obtain st' where 4: "stmt s0 ev cd (st⦇gas := g⦈) = Normal ((),
st')" by (simp split:result.split_asm prod.split_asm)
    moreover from assms 1 2 3 4 True obtain st'' where 5: "stmt (WHILE ex s0) ev cd st' = Normal ((),
st'')" by (simp split:result.split_asm prod.split_asm)
    ultimately show ?thesis using 1 2 3 that(1) assms by simp
  next
    case False
    then show ?thesis using 1 2 3 that(2) assms true_neq_false by simp
  next
    case None
    then show ?thesis using 1 2 3 assms by simp
  qed
qed

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 -
from *assms* **have** *1: "gas st > costs (INVOKE i xe) ev cd st"* **by** *(simp split:if_split_asm)*
moreover from *assms 1* **obtain** *ct fb* **where** *2: "ep $$ (contract ev) = Some (ct, fb)"* **by** *(simp split: prod.split_asm result.split_asm option.split_asm)*
moreover from *assms 1 2* **obtain** *fp f* **where** *3: "ct $$ i = Some (Method (fp, False, f))"* **by** *(simp split: prod.split_asm result.split_asm option.split_asm Member.split_asm bool.split_asm)*
moreover from *assms 1 2 3* **obtain** *e_l cd_l k_l m_l g* **where** *4: "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)"* **by** *(simp split: prod.split_asm result.split_asm)*
moreover from *assms 1 2 3 4* **obtain** *st''* **where** *5: "stmt f e_l cd_l (st⦇gas:= g, stack:=k_l, memory:=m_l⦈) = Normal ((), st'')"* **by** *(simp split: prod.split_asm result.split_asm)*
moreover from *assms 1 2 3 4 5* **have** *"st' = st''⦇stack:=stack st⦈"* **by** *(simp split: prod.split_asm result.split_asm)*
ultimately show *?thesis* **using** *that* **by** *simp*
qed

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 -
from *assms* **have** *1: "gas st > costs (EXTERNAL ad' i xe val) ev cd st"* **by** *(simp split:if_split_asm)*

moreover from *assms 1* **obtain** *adv g* **where** *2: "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)"* **by** *(simp split: prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm)*
moreover from *assms 1 2* **obtain** *c* **where** *3: "type (accounts (st⦇gas := g⦈) adv) = Some (Contract c)"* **by** *(simp split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm atype.split_asm)*
moreover from *assms 1 2 3* **obtain** *ct cn fb'* **where** *4: "ep $$ c = Some (ct, cn, fb')"* **by** *(simp add: Let_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm)*
moreover from *assms 1 2 3 4* **obtain** *v t g'* **where** *5: "expr val ev cd (st⦇gas := g⦈) g = Normal ((KValue v, Value t), g')"* **using** *1 2* **by** *(simp split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm)*
moreover from *assms 1 2 3 4 5* **have** *6: "adv ≠ address ev"* **by** *(simp add: Let_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm)*
moreover from *assms 1 2 3 4 5 6* **obtain** *v'* **where** *7: "convert t (TUInt 256) v = Some v'"* **by** *(simp add: Let_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm)*
ultimately consider *(Some) fp f* **where** *"ct $$ i = Some (Method (fp, True, f))" | (None) "fmlookup ct i = None"* **using** *assms* **by** *(simp add: Let_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm Member.split_asm bool.split_asm)*
then show *?thesis*
proof *cases*
case *(Some fp f)*
moreover from *assms 1 2 3 4 5 6 7 Some* **obtain** e_l cd_l k_l m_l *g''* **where** *8: "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'')"* **by** *(simp add: Let_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm Member.split_asm)*
moreover from *assms 1 2 3 4 5 6 7 Some 8* **obtain** *acc* **where** *9: "transfer (address ev) adv v' (accounts st) = Some acc"* **by** *(simp add: Let_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm Member.split_asm)*
moreover from *assms 1 2 3 4 5 6 7 Some 8 9* **obtain** *st''* **where** *10: "stmt f e_l cd_l (st⦇gas := g'', accounts := acc, stack:=k_l, memory:=m_l⦈) = Normal ((), st'')"* **by** *(simp add: Let_def transfer_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm Member.split_asm)*
moreover from *assms 1 2 3 4 5 6 7 Some 8 9 10* **have** *"st' = st''⦇stack:=stack st, memory := memory st⦈"* **by** *(simp add: Let_def transfer_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm Member.split_asm)*
ultimately show *?thesis* **using** *1 2 3 4 5 6 7 that(1)* **by** *simp*
next
case *None*
moreover from *assms 1 2 3 4 5 6 7 None* **obtain** *acc* **where** *8: "transfer (address ev) adv v' (accounts st) = Some acc"* **by** *(simp add: Let_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm Member.split_asm)*
moreover from *assms 1 2 3 4 5 6 7 None 8* **obtain** *st''* **where** *9: "stmt fb' (e' ct adv c v') emptyStore (st⦇gas := g', accounts := acc, stack:=emptyStore, memory:=emptyStore⦈) = Normal ((), st'')"* **by** *(simp add: Let_def transfer_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm Member.split_asm)*
moreover from *assms 1 2 3 4 5 6 7 None 8 9* **have** *"st' = st''⦇stack:=stack st, memory := memory st⦈"* **by** *(simp add: Let_def transfer_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm Member.split_asm)*
ultimately show *?thesis* **using** *1 2 3 4 5 6 7 that(2)* **by** *simp*
qed
qed

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 -
  from assms have 1: "gas st > costs (TRANSFER ad ex) ev cd st" by (simp split:if_split_asm)
  moreover from assms 1 obtain adv g where 2: "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)" by (simp add: Let_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm
Type.split_asm Types.split_asm)
  moreover from assms 1 2 obtain v t g' where 3: "expr ex ev cd (st⦇gas := g⦈) g = Normal
((KValue v, Value t), g')" by (simp add: Let_def split: if_split_asm prod.split_asm result.split_asm
Stackvalue.split_asm Type.split_asm Types.split_asm)
  moreover from assms 1 2 3 obtain v' where 4: "convert t (TUInt 256) v = Some v'" by (simp add:
Let_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm
Types.split_asm option.split_asm)
  ultimately consider (Contract) c where "type (accounts (st⦇gas := g'⦈) adv) = Some (Contract c)"
| (EOA) "type (accounts (st⦇gas := g'⦈) adv) = Some EOA" using assms by (simp add: Let_def split:
if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm
option.split_asm Member.split_asm atype.split_asm)
  then show ?thesis
  proof cases
    case (Contract c)
    moreover from assms 1 2 3 4 Contract obtain ct cn f where 5: "ep $$ c = Some (ct, cn, f)"
by (simp add: Let_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm
Type.split_asm Types.split_asm option.split_asm atype.split_asm atype.split_asm)
    moreover from assms 1 2 3 4 Contract 5 obtain acc where 6: "transfer (address ev) adv v'
(accounts st) = Some acc" by (simp add: Let_def split: if_split_asm prod.split_asm result.split_asm
Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm Member.split_asm)
    moreover from assms 1 2 3 4 Contract 5 6 obtain st'' where 7: "stmt f (e' ct adv c (st⦇gas
:= g'⦈) v') emptyStore (st⦇gas := g', accounts := acc, stack:=emptyStore, memory:=emptyStore⦈)
= Normal ((), st'')" by (simp add: Let_def split: if_split_asm prod.split_asm result.split_asm
Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm Member.split_asm)
    moreover from assms 1 2 3 4 Contract 5 6 7 have "st' = st''⦇stack:=stack st, memory := memory
st⦈" by (simp add: Let_def split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm
Type.split_asm Types.split_asm option.split_asm Member.split_asm)
    ultimately show ?thesis using 1 2 3 4 that(1) by simp
  next
    case EOA
    moreover from assms 1 2 3 4 EOA obtain acc where 5: "transfer (address ev) adv v' (accounts
st) = Some acc" by (simp add: Let_def split: if_split_asm prod.split_asm result.split_asm
Stackvalue.split_asm Type.split_asm Types.split_asm option.split_asm Member.split_asm)
    moreover from assms 1 2 3 4 EOA 5 have "st' = st⦇gas:=g', accounts:=acc⦈" by (simp add: Let_def
split: if_split_asm prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm
option.split_asm Member.split_asm)
    ultimately show ?thesis using 1 2 3 4 that(2) by simp
  qed
qed

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')"*
using *assms* **by** *(simp split:if_split_asm prod.split_asm option.split_asm)*

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')"*
using *assms* **by** *(auto split:if_split_asm result.split_asm prod.split_asm option.split_asm)*

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 -
from *assms* **have** *1: "gas st > costs (NEW i xe val) ev cd st"* **by** *(simp split:if_split_asm)*
moreover from *st0_def assms 1* **have** *2: "type (accounts st adv0) = None"* **by** *(simp split: if_split_asm)*
moreover from *st0_def assms 1 2* **obtain** *v t g* **where** *3: "expr val ev cd st0 (gas st0) = Normal((KValue v, Value t),g)"* **by** *(simp split: prod.split_asm result.split_asm Stackvalue.split_asm Type.split_asm)*
moreover from *assms 1 st0_def 2 3* **obtain** *ct cn fb* **where** *4: "ep $$ i = Some(ct, cn, fb)"* **by** *(simp split: prod.split_asm result.split_asm option.split_asm)*
moreover from *st0_def adv0_def e'_def assms 1 2 3 4* **obtain** *e$_l$ cd$_l$ k$_l$ m$_l$ g'* **where** *5: "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')"* **by** *(simp add:Let_def split: prod.split_asm result.split_asm option.split_asm)*
moreover from *st0_def adv0_def e'_def assms 1 2 3 4 5* **obtain** *acc* **where** *6: "transfer (address ev) adv0 v (accounts (st1 g')) = Some acc"* **by** *(simp add:Let_def split: prod.split_asm result.split_asm option.split_asm)*
moreover from *st0_def st1_def adv0_def e'_def assms 1 2 3 4 5 6* **obtain** *st''* **where** *"stmt (snd cn) e$_l$ cd$_l$ (st1 g'(|accounts := acc, stack:=k$_l$, memory:=m$_l$|)) = Normal ((), st'')"* **by** *(simp add:Let_def split: prod.split_asm result.split_asm option.split_asm)*
ultimately show *?thesis* **using** *that assms* **by** *simp*
qed

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"
using assms
proof (induction arbitrary: st' rule: stmt.induct)
  case (1 e cd st)
  then show ?case using skip[OF 1(1)] by auto
next
  case (2 lv ex env cd st)
  show ?case by (cases rule: assign[OF 2(1)]; simp add: 2(2))
next
  case (3 s1 s2 e cd st)
  show ?case
  proof (cases rule: comp[OF 3(3)])
    case (1 st'')
    then show ?thesis using 3 by simp
  qed
next
  case (4 ex s1 s2 e cd st)
  show ?case
  proof (cases rule: ite[OF 4(3)])
    case (1 g)
    then show ?thesis using 4 by simp
  next
    case (2 g)
    then show ?thesis using 4 by (simp split: if_split_asm)
  qed
next
  case (5 ex s0 e cd st)
  show ?case
  proof (cases rule: while[OF 5(3)])
    case (1 g st'')
    then show ?thesis using 5 by simp
  next
    case (2 g)
    then show ?thesis using 5 by simp
  qed
next
  case (6 i xe e cd st)
  show ?case
  proof (cases rule: invoke[OF 6(2)])
    case (1 ct fb fp f e_l cd_l k_l m_l g st'')
    then show ?thesis using 6 by simp
  qed
next
  case (7 ad' i xe val e cd st)
  show ?case
  proof (cases rule: external[OF 7(3)])
    case (1 adv c g ct cn fb' v t g' v' fp f e_l cd_l k_l m_l g'' acc st'')
    moreover from 7(4) have "type (acc ad) = Some ctype" using transfer_type_same[OF 1(10)] by simp
    ultimately show ?thesis using 7(1) by simp
  next
    case (2 adv c g ct cn fb' v t g' v' acc st'')
    moreover from 7(4) have "type (acc ad) = Some ctype" using transfer_type_same[OF 2(9)] by simp
    ultimately show ?thesis using 7(2) by simp
  qed
next
  case (8 ad' ex e cd st)
  show ?case
  proof (cases rule: transfer[OF 8(2)])
    case (1 v t g adv c g' v' acc ct cn f st'')
    moreover from 8(3) have "type (acc ad) = Some ctype" using transfer_type_same[OF 1(7)] by simp
    ultimately show ?thesis using 8(1) by simp
  next
    case (2 v t g adv g' v' acc)
```

```
    moreover from 8(3) have "type (acc ad) = Some ctype" using transfer_type_same[OF 2(6)] by simp
    ultimately show ?thesis by simp
  qed
next
  case (9 id0 tp s e_v cd st)
  show ?case
  proof (cases rule: blockNone[OF 9(2)])
    case (1 cd' mem' sck' e')
    then show ?thesis using 9 by simp
  qed
next
  case (10 id0 tp ex' s e_v cd st)
  show ?case
  proof (cases rule: blockSome[OF 10(2)])
    case (1 v t g cd' mem' sck' e')
    then show ?thesis using 10 by simp
  qed
next
  case (11 i xe val e cd st)
  show ?case
  proof (cases rule: new[OF 11(2)])
    case (1 v t g ct cn fb e_l cd_l k_l m_l g' acc st'')
    moreover have "hash (address e) ⌊contracts (accounts st (address e))⌋ ≠ ad" using 11(3) 1(2) by
auto
    ultimately show ?thesis
      using 11 transfer_type_same[OF 1(6)] incrementAccountContracts_type by simp
  qed
qed

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
  by unfold_locales auto
```

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)"  by Solidity_Symbex.solidity_symbex

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)" by Solidity_Symbex.solidity_symbex

lemma "msel True (MTArray 5 (MTArray 6 (MTValue TBool))) (STR ''2'') [UINT 8 5] eempty emptyStore
(mystate(|gas:=1,memory:=mymemory2|)) 1
= Exception (Err)" by Solidity_Symbex.solidity_symbex

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))"
  by (metis String.implode_explode_eq plus_literal.rep_eq)


lemma string_literal_conv: "(map String.ascii_of y = y) ⟹  (x = String.implode y) = (String.explode
x = y) "
  by auto

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''"
  by(eval)

lemma "STR ''-92134039538802366542421159375273829975''
      = createSInt 128 456484831356494564654654521238948945546546546546546999465"
  by(eval)
```

```
lemma "STR ''-128'' = createSInt 8 (-128)"
  by(eval)

lemma "STR ''244'' = (createUInt 8 500)"
  by(eval)

lemma "STR ''220443428915524155977936330922349307608''
      = (createUInt 128 456484831356494564654654521238948945546546546546546999465446546546546168)"
  by(eval)

lemma "less (TUInt 144) (TSInt 160) (STR ''5'') (STR ''8'') = Some(STR ''True'', TBool) "
  by(eval)
```

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 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›
  apply(rule ext)
  by (simp add: sorted_list_of_set'_def sorted_list_of_set_def sorted_key_list_of_set_def)

lemma sorted_list_of_set_sort_remdups' [code]:
  ‹sorted_list_of_set' (set xs) = sort (remdups xs)›
  using sorted_list_of_fset'_def' sorted_list_of_set_sort_remdups
  by metis

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 (Show_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
  by unfold_locales auto

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''"
  by(eval)

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''"
  by(eval)

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''"
  by(eval)

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''"
  by(eval)

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 ((-96834026877269277170645294669272226))]])]
  = STR ''a_s120_21_m8[0][0]==34710450786406435909527559028938314 2↩a_s120_21_m8[0][1]==56583169929733139948967092
balance==100 - EOA''"
  by(eval)

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''"
  by(eval)

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''"
  by(eval)

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
  by unfold_locales auto

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↩)''"
  by (eval)
```

```
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()''"
  by eval

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()''"
  by eval

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
  by unfold_locales auto

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''"
  by (eval)

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
  by unfold_locales auto

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''"
  by (eval)

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
  by unfold_locales auto

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()''"
  by (eval)

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
  by unfold_locales auto

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()''"
  by (eval)

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
  by unfold_locales auto

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''"
  by (eval)

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
  by unfold_locales auto
```

```
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()''"
  by (eval)
```

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)"  by Solidity_Symbex.solidity_symbex

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)" by Solidity_Symbex.solidity_symbex

lemma "msel True (MTArray 5 (MTArray 6 (MTValue TBool))) (STR ''2'') [UINT 8 5] eempty emptyStore
(mystate(|gas:=1,memory:=mymemory2|)) 1
= Exception (Err)" by Solidity_Symbex.solidity_symbex
```

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|)) >
    by(solidity_symbex)

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"
using assms by (simp add: Stackvalue.split[of "λx. wp x P E s"])

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"
using assms by (simp add: MTypes.split[of "λx. wp x P E s"])

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"
using assms by (simp add: STypes.split[of "λx. wp x P E s"])

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"
  using assms by (simp add: Type.split[of "λx. wp x P E s"])

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"
using assms by (simp add: Types.split[of "λx. wp x P E s"])

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"
using assms by (simp add: LType.split[of "λx. wp x P E s"])

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"
```

using *assms* **by** *(simp add:Denvalue.split[of "λx. wp x P E s" f g a])*

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"*
unfolding *validS_def*
using *assms* **unfolding** *wpS_def wp_def* **by** *simp*

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"*
unfolding *wpS_def wp_def* **using** *assms* **unfolding** *validS_def* **by** *simp*

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"*
apply *(subst stmt.simps(1))*
unfolding *wpS_def*
apply *wp*
using *assms* **by** *auto*

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"
  apply (subst stmt.simps(2))
  unfolding wpS_def
  apply wp
  apply (simp_all add: Ex.induct[OF assms(18,19)])
proof -
  fix a g aa b v t ab ga ac ba l t' v'
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
     and *: "local.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 "a = (KValue v, Value t)"
     and "aa = KValue v"
     and "b = Value t"
     and **: "local.lexp lv ev cd (st(|gas := g|)) g = Normal ((LStackloc l, Value t'), ga)"
     and "ab = (LStackloc l, Value t')"
     and "ac = LStackloc l"
     and "ba = Value t'"
     and "convert t t' v = Some v'"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st(|gas := ga, stack := updateStore l (KValue v') (stack st)|))"
using assms(2) unfolding ngas_def by simp
next
  fix a g aa b v t ab ga ac ba l MTypes t' v'
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
     and *: "local.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 "a = (KValue v, Value t)"
     and "aa = KValue v"
     and "b = Value t"
     and **: "local.lexp lv ev cd (st(|gas := g|)) g = Normal ((LMemloc l, Memory (MTValue t')), ga)"
     and "ab = (LMemloc l, Memory (MTValue t'))"
     and "ac = LMemloc l"
     and "ba = Memory (MTValue t')"
     and "MTypes = MTValue t'"
     and "convert t t' v = Some v'"
```

```
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st⦇gas := ga, memory := updateStore l (MValue v') (memory st)⦈)"
using assms(4) unfolding ngas_def by simp
next
  fix a g aa b v t ab ga ac ba l ta Types v'
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
     and *: "local.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 "a = (KValue v, Value t)"
     and "aa = KValue v"
     and "b = Value t"
     and **: "local.lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LStoreloc l, Storage (STValue Types)),
ga)"
     and "ab = (LStoreloc l, Storage (STValue Types))"
     and "ac = LStoreloc l"
     and "ba = Storage (STValue Types)"
     and "ta = STValue Types"
     and "convert t Types v = Some v'"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st⦇gas := ga, storage := (storage st) (address ev := fmupd l v'
(storage st (address ev)))⦈)" using assms(3) unfolding ngas_def by simp
next
  fix a g aa b p MTypes x t ab ga ac xa l MTypesa y literal ya
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
     and *: "local.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 "a = (KCDptr p, Calldata (MTArray x t))"
     and "aa = KCDptr p"
     and "b = Calldata (MTArray x t)"
     and "MTypes = MTArray x t"
     and **: "local.lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LStackloc l, Memory MTypesa), ga)"
     and "ab = (LStackloc l, Memory MTypesa)"
     and "ac = LStackloc l"
     and "xa = Memory MTypesa"
     and "accessStore l (stack st) = Some (KMemptr literal)"
     and "y = KMemptr literal"
     and "cpm2m p literal x t cd (memory st) = Some ya"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st⦇gas := ga, memory := ya⦈)" using assms(5) unfolding ngas_def
by simp
next
     fix a g aa b p MTypes x t ab ga ac xa l ta y literal ya
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
     and *: "local.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 "a = (KCDptr p, Calldata (MTArray x t))"
     and "aa = KCDptr p"
     and "b = Calldata (MTArray x t)"
     and "MTypes = MTArray x t"
```

```
    and **: "local.lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LStackloc l, Storage ta), ga)"
    and "ab = (LStackloc l, Storage ta)"
    and "ac = LStackloc l"
    and "xa = Storage ta"
    and "accessStore l (stack st) = Some (KStoptr literal)"
    and "y = KStoptr literal"
    and "cpm2s p literal x t cd (storage st (address ev)) = Some ya"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st⦇gas := ga, storage := (storage st) (address ev := ya)⦈)" using
assms(6) unfolding ngas_def by simp
next
  fix a g aa b p MTypes x t ab ga ac xa l y
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
    and *: "local.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 "a = (KCDptr p, Calldata (MTArray x t))"
    and "aa = KCDptr p"
    and "b = Calldata (MTArray x t)"
    and "MTypes = MTArray x t"
    and **: "local.lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LMemloc l, xa), ga)"
    and "ab = (LMemloc l, xa)"
    and "ac = LMemloc l"
    and "cpm2m p l x t cd (memory st) = Some y"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st⦇gas := ga, memory := y⦈)" using assms(8) unfolding ngas_def
by simp
next
  fix a g aa b p MTypes x t ab ga ac xa l y
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
    and *: "local.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 "a = (KCDptr p, Calldata (MTArray x t))"
    and "aa = KCDptr p"
    and "b = Calldata (MTArray x t)"
    and "MTypes = MTArray x t"
    and **: "local.lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LStoreloc l, xa), ga)"
    and "ab = (LStoreloc l, xa)"
    and "ac = LStoreloc l"
    and "cpm2s p l x t cd (storage st (address ev)) = Some y"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st⦇gas := ga, storage := (storage st) (address ev := y)⦈)" using
assms(7) unfolding ngas_def by simp
next
  fix a g aa b p MTypes x t ab ga ac xa l MTypesa
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
    and *: "local.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 "a = (KMemptr p, Memory (MTArray x t))"
    and "aa = KMemptr p"
```

```
    and "b = Memory (MTArray x t)"
    and "MTypes = MTArray x t"
    and **: "local.lexp lv ev cd (st(|gas := g|)) g = Normal ((LStackloc l, Memory MTypesa), ga)"
    and "ab = (LStackloc l, Memory MTypesa)"
    and "ac = LStackloc l"
    and "xa = Memory MTypesa"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st(|gas := ga, stack := updateStore l (KMemptr p) (stack st)|))"
using assms(9) unfolding ngas_def by simp
next
  fix a g aa b p MTypes x t ab ga ac xa l ta y literal ya
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
    and *: "local.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 "a = (KMemptr p, Memory (MTArray x t))"
    and "aa = KMemptr p"
    and "b = Memory (MTArray x t)"
    and "MTypes = MTArray x t"
    and **: "local.lexp lv ev cd (st(|gas := g|)) g = Normal ((LStackloc l, Storage ta), ga)"
    and "ab = (LStackloc l, Storage ta)"
    and "ac = LStackloc l"
    and "xa = Storage ta"
    and "accessStore l (stack st) = Some (KStoptr literal)"
    and "y = KStoptr literal"
    and "cpm2s p literal x t (memory st) (storage st (address ev)) = Some ya"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st(|gas := ga, storage := (storage st) (address ev := ya)|))" using
assms(10) unfolding ngas_def by simp
next
  fix a g aa b p MTypes x t ab ga ac xa l
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
    and *: "local.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 "a = (KMemptr p, Memory (MTArray x t))"
    and "aa = KMemptr p"
    and "b = Memory (MTArray x t)"
    and "MTypes = MTArray x t"
    and **: "local.lexp lv ev cd (st(|gas := g|)) g = Normal ((LMemloc l, xa), ga)"
    and "ab = (LMemloc l, xa)"
    and "ac = LMemloc l"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st(|gas := ga, memory := updateStore l (MPointer p) (memory st)|))"
using assms(12) unfolding ngas_def by simp
next
  fix a g aa b p MTypes x t ab ga ac xa l y
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
    and *: "local.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 "a = (KMemptr p, Memory (MTArray x t))"
```

```
    and "aa = KMemptr p"
    and "b = Memory (MTArray x t)"
    and "MTypes = MTArray x t"
    and **: "local.lexp lv ev cd (st(|gas := g|)) g = Normal ((LStoreloc l, xa), ga)"
    and "ab = (LStoreloc l, xa)"
    and "ac = LStoreloc l"
    and "cpm2s p l x t (memory st) (storage st (address ev)) = Some y"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st(|gas := ga, storage := (storage st) (address ev := y)|))" using
assms(11) unfolding ngas_def by simp
next
  fix a g aa b p t x ta ab ga ac xa l MTypes y literal ya
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
    and *: "local.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 ta)), g)"
    and "a = (KStoptr p, Storage (STArray x ta))"
    and "aa = KStoptr p"
    and "b = Storage (STArray x ta)"
    and "t = STArray x ta"
    and **: "local.lexp lv ev cd (st(|gas := g|)) g = Normal ((LStackloc l, Memory MTypes), ga)"
    and "ab = (LStackloc l, Memory MTypes)"
    and "ac = LStackloc l"
    and "xa = Memory MTypes"
    and "accessStore l (stack st) = Some (KMemptr literal)"
    and "y = KMemptr literal"
    and "cps2m p literal x ta (storage st (address ev)) (memory st) = Some ya"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st(|gas := ga, memory := ya|))" using assms(13) unfolding ngas_def
by simp
next
 fix a g aa b p t x ta ab ga ac xa l tb
 assume "costs (ASSIGN lv ex) ev cd st < gas st"
    and *: "local.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 ta)), g)"
    and "a = (KStoptr p, Storage (STArray x ta))"
    and "aa = KStoptr p"
    and "b = Storage (STArray x ta)"
    and "t = STArray x ta"
    and **: "local.lexp lv ev cd (st(|gas := g|)) g = Normal ((LStackloc l, Storage tb), ga)"
    and "ab = (LStackloc l, Storage tb)"
    and "ac = LStackloc l"
    and "xa = Storage tb"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st(|gas := ga, stack := updateStore l (KStoptr p) (stack st)|))"
using assms(14) unfolding ngas_def by simp
next
 fix a g aa b p t x ta ab ga ac xa l y
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
    and *: "local.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 ta)), g)"
    and "a = (KStoptr p, Storage (STArray x ta))"
    and "aa = KStoptr p"
    and "b = Storage (STArray x ta)"
    and "t = STArray x ta"
    and **: "local.lexp lv ev cd (st(|gas := g|)) g = Normal ((LMemloc l, xa), ga)"
    and "ab = (LMemloc l, xa)"
    and "ac = LMemloc l"
    and "cps2m p l x ta (storage st (address ev)) (memory st) = Some y"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st(|gas := ga, memory := y|))" using assms(16) unfolding ngas_def
by simp
next
 fix a g aa b p t x ta ab ga ac xa l y
  assume "costs (ASSIGN lv ex) ev cd st < gas st"
    and *: "local.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 ta)), g)"
    and "a = (KStoptr p, Storage (STArray x ta))"
    and "aa = KStoptr p"
    and "b = Storage (STArray x ta)"
    and "t = STArray x ta"
    and **: "local.lexp lv ev cd (st(|gas := g|)) g = Normal ((LStoreloc l, xa), ga)"
    and "ab = (LStoreloc l, xa)"
    and "ac = LStoreloc l"
    and "copy p l x ta (storage st (address ev)) = Some y"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st(|gas := ga, storage := (storage st) (address ev := y)|))" using
assms(15) unfolding ngas_def by simp
next
 fix a g aa b p t ta t' ab ga ac x l
  assume "   costs (ASSIGN lv ex) ev cd st < gas st"
    and *: "local.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 ta t')), g)"
    and "a = (KStoptr p, Storage (STMap ta t'))"
    and "aa = KStoptr p"
    and "b = Storage (STMap ta t')"
    and "t = STMap ta t'"
    and **: "local.lexp lv ev cd (st(|gas := g|)) g = Normal ((LStackloc l, x), ga)"
    and "ab = (LStackloc l, x)"
    and "ac = LStackloc l"
  moreover have "ga ≤ gas st"
  proof -
    have "ga ≤ g" using lexp_gas[OF **] by simp
    also have "... ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    finally show ?thesis .
  qed
  ultimately show "ga ≤ gas st ∧ P (st(|gas := ga, stack := updateStore l (KStoptr p) (stack st)|))"
using assms(17) unfolding ngas_def by simp
qed
```

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"
  apply (subst stmt.simps(3))
  unfolding wpS_def
  apply wp
  using assms unfolding wpS_def wp_def by (auto split:result.split)
```

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"
  apply (subst stmt.simps(4))
  unfolding wpS_def
  apply wp
  apply (simp_all add: Ex.induct[OF assms(3,4)])
  proof -
    fix a g aa ba b v
    assume "costs (ITE ex s1 s2) ev cd st < gas st"
       and *: "local.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 ⌈True⌉, Value TBool), g)"
       and "a = (KValue ⌈True⌉, Value TBool)"
       and "aa = KValue ⌈True⌉" and "ba = Value TBool" and "v = TBool" and "b = ⌈True⌉"
    then have "wpS (stmt s1 ev cd) P E (st⦇gas := g⦈)" using assms(1) by simp
    moreover have "g ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    ultimately show "wp (local.stmt s1 ev cd) (λ_ st'. gas st' ≤ gas st ∧ P st') E (st⦇gas := g⦈)"
      unfolding wpS_def wp_def by (simp split:result.split_asm prod.split_asm)
  next
    fix a g aa ba b v
    assume "costs (ITE ex s1 s2) ev cd st < gas st"
       and *: "local.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 ⌈False⌉, Value TBool), g)"
       and "a = (KValue ⌈False⌉, Value TBool)"
       and "aa = KValue ⌈False⌉" and "ba = Value TBool" and "v = TBool" and "b = ⌈False⌉"
    then have "wpS (stmt s2 ev cd) P E (st⦇gas := g⦈)" using assms(2) by simp
    moreover have "g ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF *] by simp
    ultimately show "wp (local.stmt s2 ev cd) (λ_ st'. gas st' ≤ gas st ∧ P st') E (st⦇gas := g⦈)"
      unfolding wpS_def wp_def by (simp split:result.split_asm prod.split_asm)
  qed
```

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 (induction st rule:gas_induct)
  case (1 st)
  show ?case
```

```
  unfolding wpS_def wp_def
  proof (split result.split, rule conjI; rule allI; rule impI)
    fix x1 assume *: "local.stmt (WHILE ex sm) ev cd st = Normal x1"
    then obtain b g where **: "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 b, Value TBool), g)" by (simp only:
stmt.simps, simp split:if_split_asm result.split_asm prod.split_asm Stackvalue.split_asm Type.split_asm
Types.split_asm)
    with * consider (t) "b = ShowL_bool True" | (f) "b = ShowL_bool False" by (simp add:stmt.simps
split:if_split_asm result.split_asm prod.split_asm Stackvalue.split_asm Type.split_asm Types.split_asm)
    then show "iv (accounts st, stack st, memory st, storage st) ⟶ (case x1 of (r, s') ⇒ gas s' ≤
gas st ∧ P s')"
    proof cases
      case t
      then obtain st' where ***: "stmt sm ev cd (st(|gas := g|)) = Normal ((), st')" using * ** by
(auto simp add:stmt.simps split:if_split_asm result.split_asm)
      then have ****: "local.stmt (WHILE ex sm) ev cd st' = Normal x1" using * ** t by (simp
add:stmt.simps split:if_split_asm)
      show ?thesis
      proof
        assume "iv (accounts st, stack st, memory st, storage st)"
        then have "wpS (local.stmt sm ev cd) (λst. iv (accounts st, stack st, memory st, storage st)) E
(st(|gas:=g|))" using assms(2) ** t by auto
        then have "iv (accounts st', stack st', memory st', storage st')" unfolding wpS_def wp_def
using *** by (simp split:result.split_asm)+
        moreover have "gas st ≥ costs (WHILE ex sm) ev cd st" using * by (simp add:stmt.simps
split:if_split_asm)
        then have "gas st' < gas st" using stmt_gas[OF ***] msel_ssel_expr_load_rexp_gas(3)[OF **]
while_not_zero[of ex sm ev cd st] by simp
        ultimately have "wpS (local.stmt (WHILE ex sm) ev cd) P E st'" using 1 by simp
        then show "(case x1 of (r, s') ⇒ gas s' ≤ gas st ∧ P s')" unfolding wpS_def wp_def using
**** `gas st' < gas st` by auto
      qed
    next
      case f
      show ?thesis
      proof
        assume "iv (accounts st, stack st, memory st, storage st)"
        then have "P (st(|gas := g|))" using ** assms(1) f by simp
        moreover have "x1 = ((),st(|gas := g|))" using * ** f by (simp add:stmt.simps
true_neq_false[symmetric] split:if_split_asm)
        moreover have "g ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF **] by simp
        ultimately show "case x1 of (r, s') ⇒ gas s' ≤ gas st ∧ P s'" by (auto split:prod.split)
      qed
    qed
  next
    fix x2
    show "iv (accounts st, stack st, memory st, storage st) ⟶ E x2" using assms(3,4) Ex.nchotomy by
metis
  qed
qed
```

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"
```

```
  unfolding wpS_def wp_def
proof ((split result.split | split prod.split)+, rule conjI; (rule allI | rule impI)+)
  fix x1 x1a x2
  assume "x1 = (x1a, x2)"
    and "local.stmt (BLOCK ((id0, tp), None) stm) ev cd st = Normal x1"
  then have "local.stmt (BLOCK ((id0, tp), None) stm) ev cd st = Normal (x1a, x2)" by simp
  then show "gas x2 ≤ gas st ∧ P x2"
  proof (cases rule: blockNone)
    case (1 cd' mem' sck' e')
    then show ?thesis using assms(1)[OF 1(2)] unfolding wpS_def wp_def by auto
  qed
next
  fix e
  assume "local.stmt (BLOCK ((id0, tp), None) stm) ev cd st = Exception e"
  show "E e" using assms(2,3) by (induction rule: Ex.induct)
qed

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"
  unfolding wpS_def wp_def
proof ((split result.split | split prod.split)+, rule conjI; (rule allI | rule impI)+)
  fix x1 x1a x2
  assume "x1 = (x1a, x2)"
    and "local.stmt (BLOCK ((id0, tp), Some ex') stm) ev cd st = Normal x1"
  then have "local.stmt (BLOCK ((id0, tp), Some ex') stm) ev cd st = Normal (x1a, x2)" by simp
  then show "gas x2 ≤ gas st ∧ P x2"
  proof (cases rule: blockSome)
    case (1 v t g cd' mem' sck' e')
    moreover have "g ≤ gas st - costs (BLOCK ((id0, tp), Some ex') stm) ev cd st" using
msel_ssel_expr_load_rexp_gas(3)[OF 1(2)] by simp
    ultimately show ?thesis using assms(1)[OF 1(2)] unfolding wpS_def wp_def by auto
  qed
next
  fix e
  assume "local.stmt (BLOCK ((id0, tp), Some ex') stm) ev cd st = Exception e"
  show "E e" using assms(2,3) by (induction rule: Ex.induct)
qed

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 (induction rule:stmt.induct[where ?P="λf ev cd st. ∀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)))"])
  case (1 ev cd st)
  show ?case
  proof (rule allI, rule impI, (erule conjE)+)
    fix st' assume "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" and *: "local.stmt SKIP ev cd
st = Normal ((), st')"
    then show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" using skip[OF *] by simp
  qed
next
  case (2 lv ex ev cd st)
  show ?case
  proof (rule allI, rule impI, (erule conjE)+)
    fix st' assume "address ev ≠ ad" and "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" and
3: "stmt (ASSIGN lv ex) ev cd st = Normal ((), st')"
    then show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" by (cases rule: assign[OF 3];
simp)
  qed
next
  case (3 s1 s2 ev cd st)
  show ?case
  proof (rule allI, rule impI, (erule conjE)+)
    fix st' assume l1: "address ev ≠ ad" and l12:"type (accounts st ad) = Some (Contract cname)" and
l2: "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" and l3: "stmt (COMP s1 s2) ev cd st = Normal
((), st')" and 4:"aux1 st" and 5:"aux2 st"
    then show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))"
    proof  (cases rule: comp[OF l3])
      case (1 st'')
      moreover have *:"assert Gas (λst. costs (COMP s1 s2) ev cd st < gas st) st = Normal ((), st)"
using l3 by (simp add:stmt.simps split:if_split_asm)
      moreover from l2 have "iv (storage (st⦇gas := gas st - costs (COMP s1 s2) ev cd st⦈) ad)
(ReadL_int (bal (accounts (st⦇gas := gas st - costs (COMP s1 s2) ev cd st⦈) ad)))" by simp
      moreover have **:"gas (st⦇gas:= gas st - costs (COMP s1 s2) ev cd st⦈) ≤ gas st" by simp
      then have "aux1 (st⦇gas:= gas st - costs (COMP s1 s2) ev cd st⦈)" using 4 unfolding aux1_def
using all_gas_less[OF _ **,of "λst. Qe ad iv st"] by blast
      moreover have "aux2 (st⦇gas:= gas st - costs (COMP s1 s2) ev cd st⦈)" using 5 unfolding
aux2_def using all_gas_less[OF _ **,of "λst. Qfe ad iv st"] by blast
      ultimately have "iv (storage st'' ad) (ReadL_int (bal (accounts st'' ad)))" using 3(1) C1 l1 l12
by auto
      moreover from l12 have "type (accounts st'' ad) = Some (Contract cname)" using atype_same[OF
```

```
1(2), of ad "Contract cname"] l12 by auto
      moreover have **:"gas st'' ≤ gas st" using stmt_gas[OF 1(2)] by simp
      then have "aux1 st''" using 4 unfolding aux1_def using all_gas_less[OF _ **,of "λst. Qe ad iv
st"] by blast
      moreover have "aux2 st''" using 5 unfolding aux2_def using all_gas_less[OF _ **,of "λst. Qfe ad
iv st"] by blast
      ultimately show ?thesis using 3(2)[OF * _ 1(2), of "()"] 1(3) C1 l1 by simp
    qed
  qed
next
  case (4 ex s1 s2 ev cd st)
  show ?case
  proof (rule allI, rule impI, (erule conjE)+)
    fix st' assume l1: "address ev ≠ ad" and l12:"type (accounts st ad) = Some (Contract cname)" and
l2: "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" and l3: "stmt (ITE ex s1 s2) ev cd st =
Normal ((), st')" and l4:"aux1 st" and l5:"aux2 st"
    then show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))"
    proof (cases rule: ite[OF l3])
      case (1 g)
      moreover from l2 have "iv (storage (st(|gas :=g|)) ad) (ReadL_int (bal (accounts (st(|gas := g|))
ad)))" by simp
      moreover from l12 have "type (accounts (st(|gas:= g|)) ad) = Some (Contract cname)" using
atype_same[OF 1(3), of ad "Contract cname"] l12 by auto
      moreover have **:"gas (st(|gas:= g|)) ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF 1(2)]
by simp
      then have "aux1 (st(|gas:= g|))" using l4 unfolding aux1_def using all_gas_less[OF _ **,of "λst.
Qe ad iv st"] by blast
      moreover have "aux2 (st(|gas:= g|))" using l5 unfolding aux2_def using all_gas_less[OF _ **,of
"λst. Qfe ad iv st"] by blast
      ultimately show ?thesis using 4(1) l1 by simp
    next
      case (2 g)
      moreover from l2 have "iv (storage (st(|gas := g|)) ad) (ReadL_int (bal (accounts (st(|gas := g|))
ad)))" by simp
      moreover from l12 have "type (accounts (st(|gas:= g|)) ad) = Some (Contract cname)" using
atype_same[OF 2(3), of ad "Contract cname"] l12 by auto
      moreover have **:"gas (st(|gas:= g|)) ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF 2(2)]
by simp
      then have "aux1 (st(|gas:= g|))" using l4 unfolding aux1_def using all_gas_less[OF _ **,of "λst.
Qe ad iv st"] by blast
      moreover have "aux2 (st(|gas:= g|))" using l5 unfolding aux2_def using all_gas_less[OF _ **,of
"λst. Qfe ad iv st"] by blast
      ultimately show ?thesis using 4(2) l1 true_neq_false by simp
    qed
  qed
next
  case (5 ex s0 ev cd st)
  show ?case
  proof (rule allI, rule impI, (erule conjE)+)
    fix st' assume l1: "address ev ≠ ad" and l12:"type (accounts st ad) = Some (Contract cname)"
and l2: "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" and l3: "stmt (WHILE ex s0) ev cd st =
Normal ((), st')" and l4:"aux1 st" and l5:"aux2 st"
    then show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))"
    proof (cases rule: while[OF l3])
      case (1 g st'')
      moreover from l2 have "iv (storage (st(|gas :=g|)) ad) (ReadL_int (bal (accounts (st(|gas := g|))
ad)))" by simp
      moreover have *:"gas (st(|gas:= g|)) ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF 1(2)] by
simp
      then have "aux1 (st(|gas:= g|))" using l4 unfolding aux1_def using all_gas_less[OF _ *,of "λst.
Qe ad iv st"] by blast
      moreover have "aux2 (st(|gas:= g|))" using l5 unfolding aux2_def using all_gas_less[OF _ *,of
"λst. Qfe ad iv st"] by blast
      ultimately have "iv (storage st'' ad) (ReadL_int (bal (accounts st'' ad)))" using 5(1) l1 l12 by
```

```
simp
      moreover from l12 have "type (accounts st'' ad) = Some (Contract cname)" using atype_same[OF
1(3), of ad "Contract cname"] l12 by auto
      moreover have **:"gas st'' ≤ gas st" using stmt_gas[OF 1(3)] * by simp
      then have "aux1 st''" using l4 unfolding aux1_def using all_gas_less[OF _ **,of "λst. Qe ad iv
st"] by blast
      moreover have "aux2 st''" using l5 unfolding aux2_def using all_gas_less[OF _ **,of "λst. Qfe
ad iv st"] by blast
      ultimately show ?thesis using 5(2) 1(1,2,3,4) l1 by simp
    next
      case (2 g)
      then show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" using l2 by simp
    qed
  qed
next
  case (6 i xe ev cd st)
  show ?case
  proof (rule allI, rule impI, (erule conjE)+)
    fix st' assume 1: "address ev ≠ ad" and l12:"type (accounts st ad) = Some (Contract cname)" and
2: "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" and 3: "local.stmt (INVOKE i xe) ev cd st =
Normal ((), st')" and 4:"aux1 st" and 5:"aux2 st"
    from 3 obtain ct fb' fp f e_l cd_l k_l m_l g st''
      where l0: "costs (INVOKE i xe) ev cd st < gas st"
      and l1: "ep $$ contract ev = Some (ct, fb')"
      and l2: "ct $$ i = Some (Method (fp, False, f))"
      and l3: "load False fp xe (ffold (init ct) (emptyEnv (address ev) (contract ev) (sender ev)
(svalue ev)) (fmdom 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 l4: "stmt f e_l cd_l (st⦇gas:= g, stack:=k_l, memory:=m_l⦈) = Normal ((), st'')"
      and l5: "st' = st''⦇stack:=stack st⦈"
      using invoke by blast
    from 3 have *:"assert Gas (λst. costs (INVOKE i xe) ev cd st < gas st) st = Normal ((), st)" by
(simp add:stmt.simps split:if_split_asm)
    moreover have **: "modify (λst. st⦇gas := gas st - costs (INVOKE i xe) ev cd st⦈) st = Normal ((),
st⦇gas := gas st - costs (INVOKE i xe) ev cd st⦈)" by simp
    ultimately have "∀st'. address e_l ≠ ad ∧ iv (storage (st⦇gas := g, stack := k_l, memory := m_l⦈) ad)
(ReadL_int (bal (accounts (st⦇gas := g, stack := k_l, memory := m_l⦈) ad))) ∧ local.stmt f e_l cd_l (st⦇gas
:= g, stack := k_l, memory := m_l⦈) = Normal ((), st') ∧ aux1 (st⦇gas := g, stack := k_l, memory := m_l⦈) ∧
aux2 (st⦇gas := g, stack := k_l, memory := m_l⦈) ⟶
      iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" using 6[OF * **] l1 l2 l3 l12 by (simp
split:if_split_asm result.split_asm prod.split_asm option.split_asm)
    moreover have "address (ffold (init ct) (emptyEnv (address ev) (contract ev) (sender ev) (svalue
ev)) (fmdom ct)) = address ev" using ffold_init_ad_same[of ct "(emptyEnv (address ev) (contract ev)
(sender ev) (svalue ev))"] unfolding emptyEnv_def by simp
    with 1 have "address e_l ≠ ad" using msel_ssel_expr_load_rexp_gas(4)[OF l3] by simp
    moreover from 2 have "iv (storage (st⦇gas:= g, stack:=k_l, memory:=m_l⦈) ad) (ReadL_int (bal
(accounts (st⦇gas:= g, stack:=k_l, memory:=m_l⦈) ad)))" by simp
    moreover have *:"gas (st⦇gas := g, stack := k_l, memory := m_l⦈) ≤ gas st" using
msel_ssel_expr_load_rexp_gas(4)[OF l3] by auto
    then have "aux1 (st⦇gas:= g, stack:=k_l, memory:=m_l⦈)" using 4 unfolding aux1_def using
all_gas_less[OF _ *,of "λst. Qe ad iv st"] by blast
    moreover have *:"gas (st⦇gas := g, stack := k_l, memory := m_l⦈) ≤ gas st" using
msel_ssel_expr_load_rexp_gas(4)[OF l3] by auto
    then have "aux2 (st⦇gas := g, stack := k_l, memory := m_l⦈)" using 5 unfolding aux2_def using
all_gas_less[OF _ *,of "λst. Qfe ad iv st"] by blast
    ultimately have "iv (storage st'' ad) (ReadL_int (bal (accounts st'' ad)))" using l4 C1 by auto
    then show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" using l5  by simp
  qed
next
  case (7 ad' i xe val ev cd st)
  show ?case
  proof (rule allI, rule impI, (erule conjE)+)
    fix st' assume l1: "address ev ≠ ad"
```

```
      and l12:"type (accounts st ad) = Some (Contract cname)"
      and l2: "iv (storage st ad) (ReadL_int (bal (accounts st ad)))"
      and 3: "local.stmt (EXTERNAL ad' i xe val) ev cd st = Normal ((), st')"
      and 4: "aux1 st" and 5:"aux2 st"
    show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))"
    proof (cases rule: external[OF 3])
      case (1 adv c g ct cn fb' v t g' v' fp f e_l cd_l k_l m_l g'' acc st'')
      then show ?thesis
      proof (cases "adv = ad")
        case True
        moreover from this have "cname = c" using l12 1(4) by simp
        moreover from this have "members = ct" using C1 1(5) by simp
        moreover have "gas st ≥  costs (EXTERNAL ad' i xe val) ev cd st" using 3 by (simp
add:stmt.simps split:if_split_asm)
        then have "g'' < gas st" using msel_ssel_expr_load_rexp_gas(3)[OF 1(2)]
msel_ssel_expr_load_rexp_gas(3)[OF 1(6)] msel_ssel_expr_load_rexp_gas(4)[OF 1(9)] external_not_zero[of
ad' i xe val ev cd st] by auto
        then have "Qe ad iv (st(|gas := g'', accounts := acc, stack := k_l, memory := m_l|))" using 4 un-
folding aux1_def by simp
        moreover have "g'' ≤ gas (st(|gas := g'', accounts := acc, stack := k_l, memory := m_l|))" by
simp
        moreover from l12 have "type (acc ad) = Some (Contract cname)" using transfer_type_same[OF
1(10)] by simp
        moreover have "i |∈| fmdom members" using 1(8) ‘members = ct‘ by (simp add: fmdomI)
        moreover have "members $$ i = Some (Method (fp,True,f))" using 1(8) ‘members = ct‘ by simp
        moreover have "iv (storage st ad) (ReadL_int (bal (acc ad)) - ReadL_int v')"
        proof -
          have "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" using l2 by simp
          moreover have "ReadL_int (bal (acc ad)) = ReadL_int (bal (accounts st ad)) + ReadL_int v'" using
transfer_add[OF 1(10)] l1 True by simp
          ultimately show ?thesis by simp
        qed
        ultimately have "wpS (local.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|))" unfolding
Qe_def using l1 l12 1(2) 1(6-10) by simp
        moreover have "stmt f e_l cd_l (st(|gas := g'', accounts := acc, stack := k_l, memory := m_l|)) =
Normal ((), st'')" using 1(11) by simp
        ultimately show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" unfolding wpS_def
wp_def using 1(12) by simp
      next
        case False

        from 3 have *:"assert Gas (λst. costs (EXTERNAL ad' i xe val) ev cd st < gas st) st = Normal
((), st)" by (simp add:stmt.simps split:if_split_asm)
        moreover have **: "modify (λst. st(|gas := gas st - costs (EXTERNAL ad' i xe val) ev cd st|)) st
= Normal ((), st(|gas := gas st - costs (EXTERNAL ad' i xe val) ev cd st|))" by simp
        ultimately have "∀st'. address e_l ≠ ad ∧ type (acc ad) = Some (Contract cname) ∧ iv (storage
st ad) (ReadL_int (bal (acc ad))) ∧ local.stmt f e_l cd_l (st(|gas := g'', accounts := acc, stack := k_l,
memory := m_l|)) = Normal ((), st') ∧ aux1 (st(|gas := g'', accounts := acc, stack := k_l, memory := m_l|)) ∧
aux2 (st(|gas := g'', accounts := acc, stack := k_l, memory := m_l|)) ⟶ iv (storage st' ad) (ReadL_int (bal
(accounts st' ad)))" using 7(1)[OF * **] 1 by simp
        moreover have "address (ffold (init ct) (emptyEnv adv c (address ev) v') (fmdom ct)) = adv"
using ffold_init_ad_same[of ct "(emptyEnv adv c (address ev) v)"] unfolding emptyEnv_def by simp
        with False have "address e_l ≠ ad" using msel_ssel_expr_load_rexp_gas(4)[OF 1(9)] by simp
        moreover have "bal (acc ad) = bal ((accounts st) ad)" using transfer_eq[OF 1(10)] False l1 by
simp
        then have "iv (storage st ad) (ReadL_int (bal (acc ad)))" using l2 by simp
        moreover have "type (acc ad) = Some (Contract cname)" using transfer_type_same l12 1(10) by
simp
        moreover have *:"gas (st(|gas := g'', accounts := acc, stack := k_l, memory := m_l|)) ≤
gas st" using msel_ssel_expr_load_rexp_gas(3)[OF 1(2)] msel_ssel_expr_load_rexp_gas(3)[OF 1(6)]
msel_ssel_expr_load_rexp_gas(4)[OF 1(9)] by auto
        then have "aux1 (st(|gas := g'', accounts := acc, stack := k_l, memory := m_l|))" using 4 unfold-
ing aux1_def using all_gas_less[OF _ *,of "λst. Qe ad iv st"] by blast
```

```
        moreover have "aux2 (st(|gas := g'', accounts := acc, stack := k_l, memory := m_l|))" using 5
unfolding aux2_def using all_gas_less[OF _ *,of "λst. Qfe ad iv st"] by blast
        ultimately have "iv (storage st'' ad) (ReadL_int (bal (accounts st'' ad)))" using 1(11) by simp
        then show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" using 1(12) by simp
      qed
    next
      case (2 adv c g ct cn fb' v t g' v' acc st'')
      then show ?thesis
      proof (cases "adv = ad")
        case True
        moreover have "gas st ≥  costs (EXTERNAL ad' i xe val) ev cd st" using 3 by (simp
add:stmt.simps split:if_split_asm)
        then have "gas (st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|))
< gas st" using msel_ssel_expr_load_rexp_gas(3)[OF 2(2)] msel_ssel_expr_load_rexp_gas(3)[OF 2(6)]
external_not_zero[of ad' i xe val ev cd st] by simp
        then have "Qfe ad iv (st(|gas := g', accounts := acc, stack := emptyStore, memory :=
emptyStore|))" using 5 unfolding aux2_def by simp
        moreover have "iv (storage st ad) (ReadL_int (bal (acc ad)) - ReadL_int v')"
        proof -
          have "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" using 12 by simp
          moreover have "ReadL_int (bal (acc ad)) = ReadL_int (bal (accounts st ad)) + ReadL_int v'" using
transfer_add[OF 2(9)] 11 True by simp
          ultimately show ?thesis by simp
        qed
        moreover have "g' ≤ gas (st(|gas := g', accounts := acc, stack := emptyStore, memory :=
emptyStore|))" by simp
        moreover from 112 have "type (acc ad) = Some (Contract cname)" using transfer_type_same[OF
2(9)] by simp
        ultimately have "wpS (local.stmt fb (ffold (init members) (emptyEnv ad cname (address ev) v')
(fmdom members)) emptyStore) (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|))" unfolding Qfe_def
using 11 112 2(2) 2(6-9) by blast
        moreover have "stmt fb (ffold (init members) (emptyEnv ad cname (address ev) v') (fmdom
members)) emptyStore (st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|)) =
Normal ((), st'')"
        proof -
          from True have "cname = c" using 112 2(4) by simp
          moreover from this have "fb'=fb" using C1 2(5) by simp
          moreover from True ‘cname = c‘ have "members = ct" using C1 2(5) by simp
          ultimately show ?thesis using 2(10) True by simp
        qed
        ultimately show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" unfolding wpS_def
wp_def using 2(11) by simp
      next
        case False

        from 3 have *:"assert Gas (λst. costs (EXTERNAL ad' i xe val) ev cd st < gas st) st = Normal
((), st)" by (simp add:stmt.simps split:if_split_asm)
        then have "∀st'. address (ffold (init ct) (emptyEnv adv c (address ev) v) (fmdom ct)) ≠ ad ∧
          type (acc ad) = Some (Contract cname) ∧
          iv (storage st ad) (ReadL_int (bal (acc ad))) ∧
          local.stmt fb' (ffold (init ct) (emptyEnv adv c (address ev) v') (fmdom ct)) emptyStore
(st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|)) = Normal ((), st') ∧ aux1
(st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|)) ∧ aux2 (st(|gas := g',
accounts := acc, stack := emptyStore, memory := emptyStore|))
          ⟶ iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" using 7(2)[OF *] 2 by simp
        moreover from False have "address (ffold (init ct) (emptyEnv adv c (address ev) v') (fmdom
ct)) ≠ ad" using ffold_init_ad_same[where ?e="(|address = adv, contract = c, sender = address ev,
svalue = v, denvalue = {$$}|)" and ?e'="ffold (init ct) (emptyEnv adv c (address ev) v) (fmdom ct)"]
unfolding emptyEnv_def by simp
        moreover have "bal (acc ad) = bal ((accounts st) ad)" using transfer_eq[OF 2(9)] False 11 by
simp
        then have "iv (storage st ad) (ReadL_int (bal (acc ad)))"
          using 12 by simp
```

```
        moreover have "type (acc ad) = Some (Contract cname)" using transfer_type_same l12 2(9) by
simp
        moreover have *:"gas (st(|gas := g', accounts := acc, stack := emptyStore,
memory := emptyStore|)) ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF 2(2)]
msel_ssel_expr_load_rexp_gas(3)[OF 2(6)] by simp
        then have "aux1 (st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|))"
using 4 unfolding aux1_def using all_gas_less[OF _ *,of "λst. Qe ad iv st"] by blast
        moreover have "aux2 (st(|gas := g', accounts := acc, stack := emptyStore, memory :=
emptyStore|))" using 5 unfolding aux2_def using all_gas_less[OF _ *,of "λst. Qfe ad iv st"] by blast
        ultimately have "iv (storage st'' ad) (ReadL_int (bal (accounts st'' ad)))" using 2(10) by simp
        then show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" using 2(11) by simp
      qed
    qed
  qed
next
  case (8 ad' ex ev cd st)
  show ?case
  proof (rule allI, rule impI, (erule conjE)+)
    fix st' assume l1: "address ev ≠ ad" and l12:"type (accounts st ad) = Some (Contract cname)" and
l2: "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" and 3: "local.stmt (TRANSFER ad' ex) ev cd
st = Normal ((), st')" and 4:"aux1 st" and 5:"aux2 st"
    show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))"
    proof (cases rule: transfer[OF 3])
      case (1 v t g adv c g' v' acc ct cn f st'')
      then show ?thesis
      proof (cases "adv = ad")
        case True
        moreover have "gas st ≥  costs (TRANSFER ad' ex) ev cd st" using 3 by (simp add:stmt.simps
split:if_split_asm)
        then have "gas (st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|))
< gas st" using msel_ssel_expr_load_rexp_gas(3)[OF 1(2)] msel_ssel_expr_load_rexp_gas(3)[OF 1(3)]
transfer_not_zero[of ad' ex ev cd st] by auto
        then have "Qfe ad iv (st(|gas := g', accounts := acc, stack := emptyStore, memory :=
emptyStore|))" using 5 unfolding aux2_def by simp
        moreover have "sender (ffold (init ct) (emptyEnv adv c (address ev) v') (fmdom ct)) ≠ ad"
using l1 ffold_init_ad_same[where ?e = "(emptyEnv adv c (address ev) v')" and ?e'="ffold (init ct)
(emptyEnv adv c (address ev) v') (fmdom ct)"] unfolding emptyEnv_def by simp
        moreover have "svalue (ffold (init ct) (emptyEnv adv c (address ev) v') (fmdom ct)) = v'"
using ffold_init_ad_same[where ?e = "(emptyEnv adv c (address ev) v')" and ?e'="ffold (init ct)
(emptyEnv adv c (address ev) v') (fmdom ct)", of ct "fmdom ct"] unfolding emptyEnv_def by simp
        moreover have "iv (storage st ad) (ReadL_int (bal (acc ad)) - ReadL_int v')"
        proof -
          have "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" using l2 by simp
          moreover have "ReadL_int (bal (acc ad)) = ReadL_int (bal (accounts st ad)) + ReadL_int v'" using
transfer_add[OF 1(7)] l1 True by simp
          ultimately show ?thesis by simp
        qed
        moreover have "g' ≤ gas (st(|gas := g', accounts := acc, stack := emptyStore, memory :=
emptyStore|))" by simp
        moreover from l12 have "type (acc ad) = Some (Contract cname)" using transfer_type_same[OF
1(7)] by simp
        ultimately have "wpS (local.stmt fb (ffold (init members) (emptyEnv ad cname (address ev) v')
(fmdom members)) emptyStore) (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|))" unfolding Qfe_def
using l1 l12 1(2-7) by blast
        moreover have "stmt fb (ffold (init members) (emptyEnv ad cname (address ev) v') (fmdom
members)) emptyStore (st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|)) =
Normal ((), st'')"
        proof -
          from True have "cname = c" using l12 1(5) by simp
          moreover from this have "f=fb" using C1 1(6) by simp
          moreover from True `cname = c` have "members = ct" using C1 1(6) by simp
          ultimately show ?thesis using 1(8) True by simp
        qed
```

```
        ultimately show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" unfolding wpS_def
wp_def using 1(9) by simp
      next
        case False

        from 3 have *:"assert Gas (λst. costs (TRANSFER ad' ex) ev cd st < gas st) st = Normal ((),
st)" by (simp add:stmt.simps split:if_split_asm)
        then have "∀st'. address (ffold (init ct) (emptyEnv adv c (address ev) v) (fmdom ct)) ≠ ad ∧
          type (acc ad) = Some (Contract cname) ∧
          iv (storage st ad) (ReadL_int (bal (acc ad))) ∧
          local.stmt f (ffold (init ct) (emptyEnv adv c (address ev) v') (fmdom ct)) emptyStore (st(|gas
:= g', accounts := acc, stack := emptyStore, memory := emptyStore|)) = Normal ((), st') ∧
          aux1 (st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|)) ∧ aux2
(st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|))
          ⟶ iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" using 8(1)[OF *] 1 by simp
        moreover from False have "address (ffold (init ct) (emptyEnv adv c (address ev) v') (fmdom
ct)) ≠ ad" using ffold_init_ad_same[of ct "emptyEnv adv c (address ev) v"] unfolding emptyEnv_def by
simp
        moreover have "bal (acc ad) = bal ((accounts st) ad)" using transfer_eq[OF 1(7)] False l1 by
simp
        then have "iv (storage st ad) (ReadL_int (bal (acc ad)))"
          using l2 by simp
        moreover have "type (acc ad) = Some (Contract cname)" using transfer_type_same l12 1(7) by
simp
        moreover have *:"gas (st(|gas := g', accounts := acc, stack := emptyStore,
memory := emptyStore|)) ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF 1(2)]
msel_ssel_expr_load_rexp_gas(3)[OF 1(3)] by simp
        then have "aux1 (st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|))"
using 4 unfolding aux1_def using all_gas_less[OF _ *,of "λst. Qe ad iv st"] by blast
        moreover have "aux2 (st(|gas := g', accounts := acc, stack := emptyStore, memory :=
emptyStore|))" using 5 unfolding aux2_def using all_gas_less[OF _ *,of "λst. Qfe ad iv st"] by blast
        ultimately have "iv (storage st'' ad) (ReadL_int (bal (accounts st'' ad)))" using 1(8) C1 by
simp
        then show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" using 1(9) by simp
      qed
    next
      case (2 v t g adv g' v' acc)
      moreover from 2(5) have "adv ≠ ad" using C1 l12 by auto
      then have "bal (acc ad) = bal (accounts st ad)" using transfer_eq[OF 2(6)] l1 by simp
      ultimately show ?thesis using l2 by simp
    qed
  qed
next
  case (9 id0 tp s ev cd st)
  show ?case
  proof (rule allI, rule impI, (erule conjE)+)
    fix st' assume l1: "address ev ≠ ad" and l12:"type (accounts st ad) = Some (Contract cname)" and
l2: "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" and l3: "stmt (BLOCK ((id0, tp), None) s) ev
cd st = Normal ((), st')" and 4:"aux1 st" and 5:"aux2 st"
    then show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))"
    proof (cases rule: blockNone[OF l3])
      case (1 cd' mem' sck' e')
      moreover from l2 have "iv (storage (st(|gas := gas st - costs (BLOCK ((id0, tp), None) s) ev cd
st, stack := sck', memory := mem'|)) ad) (ReadL_int (bal (accounts (st(|gas := gas st - costs (BLOCK ((id0,
tp), None) s) ev cd st, stack := sck', memory := mem'|)) ad)))" by simp
      moreover have *:"gas (st(|gas:= gas st - costs (BLOCK ((id0, tp), None) s) ev cd st, stack :=
sck', memory := mem'|)) ≤ gas st" by simp
      then have "aux1 (st(|gas:= gas st - costs (BLOCK ((id0, tp), None) s) ev cd st, stack := sck',
memory := mem'|))" using 4 unfolding aux1_def using all_gas_less[OF _ *,of "λst. Qe ad iv st"] by blast
      moreover have "aux2 (st(|gas:= gas st - costs (BLOCK ((id0, tp), None) s) ev cd st, stack :=
sck', memory := mem'|))" using 5 unfolding aux2_def using all_gas_less[OF _ *,of "λst. Qfe ad iv st"]
by blast
      moreover have "address e' ≠ ad" using decl_env[OF 1(2)] l1 by simp
      ultimately show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" using 9(1) l12 by simp
```

```
    qed
  qed
next
  case (10 id0 tp ex' s ev cd st)
  show ?case
  proof (rule allI, rule impI, (erule conjE)+)
    fix st' assume l1: "address ev ≠ ad" and l12:"type (accounts st ad) = Some (Contract cname)" and
l2: "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" and l3: "stmt (BLOCK ((id0, tp), Some ex')
s) ev cd st = Normal ((), st')" and 4:"aux1 st" and 5:"aux2 st"
    then show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))"
    proof  (cases rule: blockSome[OF l3])
      case (1 v t g cd' mem' sck' e')
      moreover from l2 have "iv (storage (st(|gas := g, stack := sck', memory := mem'|)) ad) (ReadL_int
(bal (accounts (st(|gas := g, stack := sck', memory := mem'|)) ad)))" by simp
      moreover have *:"gas (st(|gas:= g, stack := sck', memory := mem'|)) ≤ gas st" using
msel_ssel_expr_load_rexp_gas(3)[OF 1(2)] by simp
      then have "aux1 (st(|gas:= g, stack := sck', memory := mem'|))" using 4 unfolding aux1_def using
all_gas_less[OF _ *,of "λst. Qe ad iv st"] by blast
      moreover have "aux2 (st(|gas:= g, stack := sck', memory := mem'|))" using 5 unfolding aux2_def
using all_gas_less[OF _ *,of "λst. Qfe ad iv st"] by blast
      moreover have "address e' ≠ ad" using decl_env[OF 1(3)] l1 by simp
      ultimately show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))" using 10(1) l12 by simp
    qed
  qed
next
  case (11 i xe val ev cd st)
  show ?case
  proof (rule allI, rule impI, (erule conjE)+)
    fix st' assume l1: "address ev ≠ ad" and l12:"type (accounts st ad) = Some (Contract cname)" and
l2: "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" and l3: "stmt (NEW i xe val) ev cd st =
Normal ((), st')" and 4:"aux1 st" and 5:"aux2 st"
    then show "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))"
    proof  (cases rule: new[OF l3])
      case (1 v t g ct cn fb e_l cd_l k_l m_l g' acc st'')
      moreover define adv where "adv = hash (address ev) ⌊contracts (accounts (st(|gas := gas st -
costs (NEW i xe val) ev cd st|)) (address ev))⌋"
      moreover define st''' where "st''' = (st(|gas := gas st - costs (NEW i xe val) ev cd st, gas
:= g', accounts := (accounts st)(adv := (|bal = ShowL_int 0, type = Some (Contract i), contracts = 0|)),
storage := (storage st)(adv := {$$}), accounts := acc, stack := k_l, memory := m_l|))"
      ultimately have "∀st'. address e_l ≠ ad ∧
        type (accounts st''' ad) = Some (Contract cname) ∧
        iv (storage st''' ad) ⌈bal (accounts st''' ad)⌉ ∧
        local.stmt (snd cn) e_l cd_l st''' = Normal ((), st') ∧ aux1 st''' ∧ aux2 st''' ⟶
        iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
        using 11 by simp
      moreover have "address e_l ≠ ad"
      proof -
        have "address e_l = adv" using msel_ssel_expr_load_rexp_gas(4)[OF 1(5)] adv_def by simp
        moreover have "adv ≠ ad" using l12 1(2) adv_def by auto
        ultimately show ?thesis by simp
      qed
      moreover have "type (accounts st''' ad) = Some (Contract cname)"
      proof -
        have "adv ≠ ad" using l12 1(2) adv_def by auto
        then have "type (accounts st ad) = type (acc ad)" using transfer_type_same[OF 1(6)] adv_def by
simp
        then show ?thesis using l12 st'''_def by simp
      qed
      moreover have "iv (storage st''' ad) ⌈bal (accounts st''' ad)⌉"
      proof -
        have "adv ≠ ad" using l12 1(2) adv_def by auto
        then have "bal (accounts st ad) = bal (accounts st''' ad)" using transfer_eq[OF 1(6), of ad]
l1 using st'''_def adv_def by simp
        moreover have "storage st ad = storage st''' ad" using st'''_def ‘adv ≠ ad‘ by simp
```

```
        ultimately show ?thesis using 12 by simp
      qed
      moreover have "local.stmt (snd cn) e_l cd_l st''' = Normal ((), st'')" using 1(7) st'''_def
adv_def by simp
      moreover have "aux1 st'''"
      proof -
        have *: "gas st''' ≤ gas st" unfolding st'''_def using msel_ssel_expr_load_rexp_gas(3)[OF
1(3)] msel_ssel_expr_load_rexp_gas(4)[OF 1(5)] by auto
        then show ?thesis using 4 unfolding aux1_def using all_gas_less[OF _ *,of "λst. Qe ad iv st"]
by simp
      qed
      moreover have "aux2 st'''"
      proof -
        have *: "gas st''' ≤ gas st" unfolding st'''_def using msel_ssel_expr_load_rexp_gas(3)[OF
1(3)] msel_ssel_expr_load_rexp_gas(4)[OF 1(5)] by auto
        then show ?thesis using 5 unfolding aux2_def using all_gas_less[OF _ *,of "λst. Qe ad iv st"]
by simp
      qed
      ultimately have "iv (storage st'' ad) ⌈bal (accounts st'' ad)⌉" by simp
      moreover have "storage st'' ad = storage st' ad" using 1(8) by simp
      moreover have "bal (accounts st'' ad) = bal (accounts st' ad)" using 1(8) by simp
      ultimately show ?thesis by simp
    qed
  qed
qed

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 *(induction st rule: gas_induct)*
case *(1 st)*
show *?case* **unfolding** *Pe_def Pi_def Pfi_def Pfe_def*
proof *elims*
fix *ev::Environment* **and** *ad' i xe val cd*
assume *a00: "type (accounts st ad) = Some (Contract cname)"*

```
      and a0: "address ev = ad"
      and a1: "∀adv c g v t g' v'.
         local.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 ∧
         local.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')"
    show "wpS (local.stmt (EXTERNAL ad' i xe val) ev cd) (λst. iv (storage st ad) (ReadL_int (bal
(accounts st ad)))) (λe. e = Gas ∨ e = Err) st" unfolding wpS_def wp_def
    proof (split result.split; split prod.split; rule conjI; (rule allI)+; (rule impI)+)
      fix x1 x1a s''''''
      assume "x1 = (x1a, s'''''')" and 2: "local.stmt (EXTERNAL ad' i xe val) ev cd st = Normal x1"
      then have "local.stmt (EXTERNAL ad' i xe val) ev cd st = Normal (x1a, s'''''')" by simp
      then show "gas s'''''' ≤ gas st ∧ iv (storage s'''''' ad) (ReadL_int (bal (accounts s''''''
ad)))"
      proof (cases rule: external)
        case (Some adv0 c0 g0 ct0 cn0 fb0 v0 t0 g0' v0' fp0 f0 e_l0 cd_l0 k_l0 m_l0 g0'' acc0 st0'')
        moreover have "iv (storage st0'' ad) (ReadL_int (bal (accounts st0'' ad)))"
        proof -
          from Some(3) have "adv0 ≠ ad" using a0 by simp
          then have "address e_l0 ≠ ad" using msel_ssel_expr_load_rexp_gas(4)[OF Some(9)]
ffold_init_ad_same[of ct0 "(emptyEnv adv0 c0 (address ev) v0')" "(fmdom ct0)" "(ffold (init ct0)
(emptyEnv adv0 c0 (address ev) v0) (fmdom ct0))"] unfolding emptyEnv_def by simp
          moreover have "type (accounts (st⦇gas := g0'', accounts := acc0, stack := k_l0, memory :=
m_l0⦈) ad) = Some (Contract cname)" using transfer_type_same[OF Some(10)] a00 by simp
          moreover have "iv (storage (st⦇gas := g0'', accounts := acc0, stack := k_l0, memory := m_l0⦈)
ad)
                         (ReadL_int (bal (accounts (st⦇gas := g0'', accounts := acc0, stack := k_l0, memory
:= m_l0⦈) ad)))"
          proof -
            from Some(5) have "c0 |∈| fmdom ep" by (rule fmdomI)
            with a0 a1 Some ‘adv0 ≠ ad‘ have "iv (storage st ad) (ReadL_int (bal (accounts st ad)) -
ReadL_int v0')" by simp
            moreover have "ReadL_int (bal (acc0 ad)) = ReadL_int (bal (accounts st ad)) - ReadL_int v0'"
using transfer_sub[OF Some(10)] a0 ‘adv0 ≠ ad‘ by simp
            ultimately show ?thesis by simp
          qed
          moreover have "∀st'::State. gas st' < gas (st⦇gas := g0'', accounts := acc0, stack := k_l0,
memory := m_l0⦈) ⟶
              (∀mid fp f ev.
                members $$ mid = Some (Method (fp, True, f)) ∧
                address ev ≠ ad
               ⟶ (∀adex cd st0 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) ∧
                    local.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) ∧
                    local.expr val ev cd (st0⦇gas := g⦈) g = Normal ((KValue v, Value t), g') ∧
                    convert t (TUInt 256) v = Some v' ∧
                    local.load True fp 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'') ∧
                    transfer (address ev) ad v' (accounts (st0⦇gas := g''⦈)) = Some acc ∧
                    iv (storage st0 ad) (ReadL_int (bal (acc ad)) - ReadL_int v')
                   ⟶ wpS (local.stmt f 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'⦈))))" (is
"∀st'. ?left st' ⟶ ?right st'")
          proof (rule allI,rule impI)
            fix st'::State
            assume "gas st' < gas (st⦇gas := g0'', accounts := acc0, stack := k_l0, memory := m_l0⦈)"
```

```
          then have "gas st' < gas st" using msel_ssel_expr_load_rexp_gas(4)[OF Some(9)]
msel_ssel_expr_load_rexp_gas(3)[OF Some(2)] msel_ssel_expr_load_rexp_gas(3)[OF Some(6)] by auto
          then show "?right st'" using assm[OF all_gas_le[OF `gas st' < gas st` "1.IH"], THEN
conjunct1] unfolding Qe_def by simp
        qed
        moreover have "∀st'::State. gas st' < gas (st⦇gas := g0'', accounts := acc0, stack := k_l0,
memory := m_l0⦈) ⟶
            (∀ev. address ev ≠ ad
             ⟶ (∀ex cd st0 adex cc v t g g' v' acc.
                  g' ≤ gas st' ∧
                  type (acc ad) = Some (Contract cname) ∧
                  expr adex ev cd (st0⦇gas := gas st0 - cc⦈) (gas st0 - cc) = Normal ((KValue ad,
Value TAddr), g) ∧
                  expr ex ev cd (st0⦇gas := g⦈) g = Normal ((KValue v, Value t), g') ∧
                  convert t (TUInt 256) v = Some v' ∧
                  transfer (address ev) ad v' (accounts st0) = Some acc ∧
                  iv (storage st0 ad) (⌈bal (acc ad)⌉ - ⌈v'⌉) ⟶
                  wpS (local.stmt fb (ffold (init members) (emptyEnv ad cname (address ev) v') (fmdom
members)) emptyStore)
                   (λst. iv (storage st ad) (ReadL_int (bal (accounts st ad)))) (λe. e = Gas ∨ e =
Err)
                   (st0⦇gas := g', accounts := acc, stack := emptyStore, memory := emptyStore⦈)))"
(is "∀st'. ?left st' ⟶ ?right st'")
        proof (rule allI,rule impI)
          fix st'::State
          assume l0: "gas st' < gas (st⦇gas := g0'', accounts := acc0, stack := k_l0, memory := m_l0⦈)"
          then have "gas st' < gas st" using msel_ssel_expr_load_rexp_gas(4)[OF Some(9)]
msel_ssel_expr_load_rexp_gas(3)[OF Some(2)] msel_ssel_expr_load_rexp_gas(3)[OF Some(6)] by auto
          then show "?right st'" using assm[OF all_gas_le[OF `gas st' < gas st` "1.IH"], THEN
conjunct2, THEN conjunct2, THEN conjunct2] unfolding Qfe_def by simp
        qed
        ultimately show ?thesis using safeStore[of e_l0 ad "st⦇gas := g0'', accounts := acc0, stack
:= k_l0, memory := m_l0⦈" iv f0 cd_l0 st0''] Some unfolding Qe_def Qfe_def by blast
      qed
      moreover have "gas st0'' ≤ gas st" using msel_ssel_expr_load_rexp_gas(4)[OF Some(9),THEN
conjunct1] msel_ssel_expr_load_rexp_gas(3)[OF Some(2)] msel_ssel_expr_load_rexp_gas(3)[OF Some(6)]
stmt_gas[OF Some(11)] by simp
      ultimately show ?thesis by simp
    next
      case (None adv0 c0 g0 ct0 cn0 fb0' v0 t0 g0' v0' acc0 st0'')
      moreover have "iv (storage s'''''' ad) (ReadL_int (bal (accounts s'''''' ad)))"
      proof -
        from None have "adv0 ≠ ad" using a0 by auto
        then have "address (ffold (init ct0) (emptyEnv adv0 c0 (address ev) v0') (fmdom ct0)) ≠ ad"
using ffold_init_ad_same[where ?e="⦇address = adv0, contract = c0, sender = address ev, svalue = v0',
denvalue = {$$}⦈" and e'="ffold (init ct0) (emptyEnv adv0 c0 (address ev) v0') (fmdom ct0)"] unfolding
emptyEnv_def by simp
        moreover have "type (accounts (st⦇gas := g0', accounts := acc0, stack := emptyStore, memory
:= emptyStore⦈) ad) = Some (Contract cname)" using transfer_type_same[OF None(9)] a00 by simp
        moreover have "iv (storage (st⦇gas := g0', accounts := acc0, stack := emptyStore, memory :=
emptyStore⦈) ad) (ReadL_int (bal (accounts (st⦇gas := g0', accounts := acc0, stack := emptyStore, memory
:= emptyStore⦈) ad)))"
        proof -
          from None(5) have "c0 |∈| fmdom ep" by (rule fmdomI)
          with a0 a1 None `adv0 ≠ ad` have "iv (storage st ad) (ReadL_int (bal (accounts st ad)) -
ReadL_int v0')" by simp
          moreover have "ReadL_int (bal (acc0 ad)) = ReadL_int (bal (accounts st ad)) - ReadL_int v0'"
using transfer_sub[OF None(9)] a0 `adv0 ≠ ad` by simp
          ultimately show ?thesis by simp
        qed
        moreover have "∀st'::State. gas st' < gas (st⦇gas := g0', accounts := acc0, stack :=
emptyStore, memory := emptyStore⦈) ⟶
            (∀mid fp f ev.
              members $$ mid = Some (Method (fp, True, f)) ∧
```

```
                address ev ≠ ad
               ⟶ (∀adex cd st0 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) ∧
                    local.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) ∧
                    local.expr val ev cd (st0(|gas := g|)) g = Normal ((KValue v, Value t), g') ∧
                    convert t (TUInt 256) v = Some v' ∧
                    local.load True fp 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'') ∧
                    transfer (address ev) ad v' (accounts (st0(|gas := g''|))) = Some acc ∧
                    iv (storage st0 ad) (ReadL_int (bal (acc ad)) - ReadL_int v')
                   ⟶  wpS (local.stmt f 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'|))))" (is
"∀st'. ?left st' ⟶ ?right st'")
          proof (rule allI,rule impI)
            fix st'::State
            assume "gas st' < gas (st(|gas := g0', accounts := acc0, stack := emptyStore, memory :=
emptyStore|))"
            then have "gas st' < gas st" using msel_ssel_expr_load_rexp_gas(3)[OF None(2)]
msel_ssel_expr_load_rexp_gas(3)[OF None(6)] by auto
            then show "?right st'" using assm[OF all_gas_le[OF `gas st' < gas st` "1.IH"], THEN
conjunct1] unfolding Qe_def by simp
          qed
          moreover have "∀st'::State. gas st' < gas (st(|gas := g0', accounts := acc0, stack :=
emptyStore, memory := emptyStore|)) ⟶
              (∀ev. address ev ≠ ad
               ⟶ (∀ex cd st0 adex cc v t g g' v' acc.
                    g' ≤ gas st' ∧
                    type (acc ad) = Some (Contract cname) ∧
                    expr adex ev cd (st0(|gas := gas st0 - cc|)) (gas st0 - cc) = Normal ((KValue ad,
Value TAddr), g) ∧
                    expr ex ev cd (st0(|gas := g|)) g = Normal ((KValue v, Value t), g') ∧
                    convert t (TUInt 256) v = Some v' ∧
                    transfer (address ev) ad v' (accounts st0) = Some acc ∧
                    iv (storage st0 ad) (⌈bal (acc ad)⌉ - ⌈v'⌉) ⟶
                    wpS (local.stmt fb (ffold (init members) (emptyEnv ad cname (address ev) v') (fmdom
members)) emptyStore)
                     (λst. iv (storage st ad) (ReadL_int (bal (accounts st ad)))) (λe. e = Gas ∨ e =
Err)
                     (st0(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|))))"
(is "∀st'. ?left st' ⟶ ?right st'")
          proof (rule allI,rule impI)
            fix st'::State
            assume 10: "gas st' < gas (st(|gas := g0', accounts := acc0, stack := emptyStore, memory :=
emptyStore|))"
            then have "gas st' < gas st" using msel_ssel_expr_load_rexp_gas(3)[OF None(2)]
msel_ssel_expr_load_rexp_gas(3)[OF None(6)] by auto
            then show "?right st'" using assm[OF all_gas_le[OF `gas st' < gas st` "1.IH"], THEN
conjunct2, THEN conjunct2, THEN conjunct2] unfolding Qfe_def by simp
          qed
          ultimately have "iv (storage st0'' ad) (ReadL_int (bal (accounts st0'' ad)))" using
safeStore[of "ffold (init ct0) (emptyEnv adv0 c0 (address ev) v0') (fmdom ct0)" ad "st(|gas := g0',
accounts := acc0, stack := emptyStore, memory := emptyStore|)" iv fb0' emptyStore "st0''"] None unfold-
ing Qe_def Qfe_def by blast
          then show ?thesis using None(11) by simp
        qed
        moreover have "gas st0'' ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF None(2)]
msel_ssel_expr_load_rexp_gas(3)[OF None(6)] stmt_gas[OF None(10)] by simp
        ultimately show ?thesis by simp
      qed
    next
      fix e
```

```
      assume "local.stmt (EXTERNAL ad' i xe val) ev cd st = Exception e"
      show "e = Gas ∨ e = Err" using Ex.nchotomy by simp
    qed
  next
    fix ev ex ad' cd
    assume a00: "type (accounts st ad) = Some (Contract cname)"
      and a0: "address ev = ad"
      and a1: "∀adv g. local.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"
      and a2: "∀adv g v t g' v'.
          local.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 ∧
          local.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')"
    show "wpS (local.stmt (TRANSFER ad' ex) ev cd) (λst. iv (storage st ad) (ReadL_int (bal (accounts st
ad)))) (λe. e = Gas ∨ e = Err) st"
      unfolding wpS_def wp_def
    proof (split result.split; split prod.split; rule conjI; (rule allI)+; (rule impI)+)
      fix x1 x1a s''''''
      assume "x1 = (x1a, s'''''')" and "local.stmt (TRANSFER ad' ex) ev cd st = Normal x1"
      then have 2: "local.stmt (TRANSFER ad' ex) ev cd st = Normal (x1a, s'''''')" by simp
      then show "gas s'''''' ≤ gas st ∧ iv (storage s'''''' ad) (ReadL_int (bal (accounts s'''''' 
ad)))"
      proof (cases rule: transfer)
        case (Contract v0 t0 g0 adv0 c0 g0' v0' acc0 ct0 cn0 f0 st0'')
        moreover have "iv (storage s'''''' ad) (ReadL_int (bal (accounts s'''''' ad)))"
        proof -
          from Contract have "adv0 ≠ ad" using a1 by auto
          then have "address (ffold (init ct0) (emptyEnv adv0 c0 (address ev) v0') (fmdom ct0)) ≠ ad"
using a0 ffold_init_ad_same[where ?e'="ffold (init ct0) (emptyEnv adv0 c0 (address ev) v0') (fmdom
ct0)"] unfolding emptyEnv_def by simp
          moreover have "type (accounts (st⦇gas := g0', accounts := acc0, stack := emptyStore, memory
:= emptyStore⦈) ad) = Some (Contract cname)" using transfer_type_same[OF Contract(7)] a00 by simp
          moreover have "iv (storage (st⦇gas := g0', accounts := acc0, stack := emptyStore, memory :=
emptyStore⦈) ad) (ReadL_int (bal (accounts (st⦇gas := g0', accounts := acc0, stack := emptyStore, memory
:= emptyStore⦈) ad)))"
          proof -
            from a0 a2 Contract ‘adv0 ≠ ad‘ have "iv (storage st ad) (ReadL_int (bal (accounts st ad))
- ReadL_int v0')" by blast
            moreover have "ReadL_int (bal (acc0 ad)) = ReadL_int (bal (accounts st ad)) - ReadL_int v0'"
using transfer_sub[OF Contract(7)] a0 ‘adv0 ≠ ad‘ by simp
            ultimately show ?thesis by simp
          qed
          moreover have "∀st'::State. gas st' < gas (st⦇gas := g0', accounts := acc0, stack :=
emptyStore, memory := emptyStore⦈) ⟶ Qe ad iv st'"
          proof (rule allI,rule impI)
            fix st'::State
            assume "gas st' < gas (st⦇gas := g0', accounts := acc0, stack := emptyStore, memory :=
emptyStore⦈)"
            then have "gas st' < gas st" using msel_ssel_expr_load_rexp_gas(3)[OF Contract(2)]
msel_ssel_expr_load_rexp_gas(3)[OF Contract(3)] by auto
            then show "Qe ad iv st'" using assm[OF all_gas_le[OF ‘gas st' < gas st‘ "1.IH"], THEN
conjunct1] by simp
          qed
          moreover have "∀st'::State. gas st' < gas (st⦇gas := g0', accounts := acc0, stack :=
emptyStore, memory := emptyStore⦈) ⟶
              (∀ev. address ev ≠ ad
               ⟶ (∀ex cd st0 adex cc v t g g' v' acc.
                     g' ≤ gas st' ∧
                     type (acc ad) = Some (Contract cname) ∧
                     expr adex ev cd (st0⦇gas := gas st0 - cc⦈) (gas st0 - cc) = Normal ((KValue ad,
Value TAddr), g) ∧
```

```
                    expr ex ev cd (st0(|gas := g|)) g = Normal ((KValue v, Value t), g') ∧
                    convert t (TUInt 256) v = Some v' ∧
                    transfer (address ev) ad v' (accounts st0) = Some acc ∧
                    iv (storage st0 ad) (⌈bal (acc ad)⌉ - ⌈v'⌉) ⟶
                    wpS (local.stmt fb (ffold (init members) (emptyEnv ad cname (address ev) v') (fmdom
members)) emptyStore)
                     (λst. iv (storage st ad) (ReadL_int (bal (accounts st ad)))) (λe. e = Gas ∨ e =
Err)
                     (st0(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|)))"
(is "∀st'. ?left st' ⟶ ?right st'")
        proof (rule allI,rule impI)
          fix st'::State
          assume l0: "gas st' < gas (st(|gas := g0', accounts := acc0, stack := emptyStore, memory :=
emptyStore|))"
          then have "gas st' < gas st" using msel_ssel_expr_load_rexp_gas(3)[OF Contract(2)]
msel_ssel_expr_load_rexp_gas(3)[OF Contract(3)] by auto
          then show "?right st'" using assm[OF all_gas_le[OF ‘gas st' < gas st‘ "1.IH"], THEN
conjunct2, THEN conjunct2, THEN conjunct2] unfolding Qfe_def by simp
        qed
        ultimately have "iv (storage st0'' ad) (ReadL_int (bal (accounts st0'' ad)))" using
safeStore[of "ffold (init ct0) (emptyEnv adv0 c0 (address ev) v0') (fmdom ct0)" ad "st(|gas := g0',
accounts := acc0, stack := emptyStore, memory := emptyStore|)" iv f0 emptyStore "st0''"] Contract un-
folding Qe_def Qfe_def by blast
        then show ?thesis using Contract(9) by simp
      qed
      moreover have "gas st0'' ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF Contract(2)]
msel_ssel_expr_load_rexp_gas(3)[OF Contract(3)] stmt_gas[OF Contract(8)] by simp
      ultimately show ?thesis by simp
    next
      case (EOA v0 t0 g0 adv0 g0' v0' acc0)
      moreover have "iv (storage (st(|gas := g0', accounts := acc0|)) ad) (ReadL_int (bal (accounts
(st(|gas := g0', accounts := acc0|)) ad)))"
      proof -
        from EOA have "adv0 ≠ ad" using a1 by auto
        with a0 a2 EOA have "iv (storage st ad) (ReadL_int (bal (accounts st ad)) - ReadL_int v0')" by
blast
        moreover have "ReadL_int (bal (acc0 ad)) = ReadL_int (bal (accounts st ad)) - ReadL_int v0'"
using transfer_sub[OF EOA(6)] a0 ‘adv0 ≠ ad‘ by simp
        ultimately show ?thesis by simp
      qed
      moreover have "g0' ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF EOA(2)]
msel_ssel_expr_load_rexp_gas(3)[OF EOA(3)] by simp
      ultimately show ?thesis by simp
    qed
  next
    fix e
    assume "local.stmt (TRANSFER ad' ex) ev cd st = Exception e"
    show "e = Gas ∨ e = Err" using Ex.nchotomy by simp
  qed
next
  fix ev i xe cd fp
  assume a0: "type (accounts st ad) = Some (Contract cname)"
     and ad_ev: "address ev = ad"
     and a1: "contract ev = cname"
     and pre: "∀fp e_l cd_l k_l m_l g.
        local.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)"
  show "wpS (local.stmt (INVOKE i xe) ev cd) (λst. post i (ReadL_int (bal (accounts st ad)), storage
st ad)) (λe. e = Gas ∨ e = Err) st"
    unfolding wpS_def wp_def
  proof (split result.split; split prod.split; rule conjI; (rule allI)+; (rule impI)+)
```

```
      fix x1 x1a st'
      assume "x1 = (x1a, st')"
         and *: "local.stmt (INVOKE i xe) ev cd st = Normal x1"
      then have "local.stmt (INVOKE i xe) ev cd st = Normal (x1a, st')" by simp
      then show "gas st' ≤ gas st ∧ post i (ReadL_int (bal (accounts st' ad)), storage st' ad)"
      proof (cases rule: invoke)
        case (1 ct fb fp f e_l cd_l k_l m_l g st'')
        have "post i (ReadL_int (bal (accounts st' ad)), storage st' ad)"
        proof -
          from * have "gas st > costs (INVOKE i xe) ev cd st" by (simp add:stmt.simps
split:if_split_asm)
          then have "gas (st(|gas := gas st - costs (INVOKE i xe) ev cd st|)) < gas st" using
invoke_not_zero[of i xe ev cd st] by simp

          from a1 have "ct = members" using 1(2) C1 by simp
          then have **: "local.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)" using 1(4) ad_ev by simp
          moreover from 1(2,3) have ***: "members $$ i = Some (Method (fp, False, f))" using ad_ev
'ct = members' by simp
          ultimately have "pre i (ReadL_int (bal (accounts st ad)), storage st ad, e_l, cd_l, k_l, m_l)"
using pre by blast
          moreover have "g ≤ gas (st(|gas := gas st - costs (INVOKE i xe) ev cd st|))" using
msel_ssel_expr_load_rexp_gas(4)[OF 1(4),THEN conjunct1] by simp
          ultimately have "wpS (local.stmt f e_l cd_l) (λst. post i (ReadL_int (bal (accounts st ad)),
storage st ad)) (λe. e = Gas ∨ e = Err)
            (st(|gas := g, stack := k_l, memory := m_l|))" using assm[OF all_gas_le[OF 'gas (st(|gas := gas
st - costs (INVOKE i xe) ev cd st|)) < gas st' "1.IH"], THEN conjunct2, THEN conjunct1] ** *** ad_ev a1
unfolding Qi_def by simp
          then show ?thesis unfolding wpS_def wp_def using 1(5,6) by simp
        qed
        moreover have "gas st' ≤ gas st" using msel_ssel_expr_load_rexp_gas(4)[OF 1(4),THEN
conjunct1] stmt_gas[OF 1(5)] 1(6) by simp
        ultimately show ?thesis by simp
      qed
    next
      fix e
      assume "local.stmt (INVOKE i xe) ev cd st = Exception e"
      show "e = Gas ∨ e = Err" using Ex.nchotomy by simp
    qed
  next
    fix ev ex ad' cd
    assume a0: "type (accounts st ad) = Some (Contract cname)"
      and ad_ev: "address ev = ad"
      and a1: "∀adv g.
          local.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"
      and a2: "∀g v t g'.
          local.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) ∧
          local.expr ex ev cd (st(|gas := g|)) g = Normal ((KValue v, Value t), g') ⟶
          pref (ReadL_int (bal (accounts st ad)), storage st ad)"
    show "wpS (local.stmt (TRANSFER ad' ex) ev cd) (λst. postf (ReadL_int (bal (accounts st ad)),
storage st ad)) (λe. e = Gas ∨ e = Err) st"
      unfolding wpS_def wp_def
    proof (split result.split; split prod.split; rule conjI; (rule allI)+; (rule impI)+)
      fix x1 x1a st'
      assume "x1 = (x1a, st')" and "local.stmt (TRANSFER ad' ex) ev cd st = Normal x1"
      then have 2: "local.stmt (TRANSFER ad' ex) ev cd st = Normal (x1a, st')" by simp
      then show "gas st' ≤ gas st ∧ postf (ReadL_int (bal (accounts st' ad)), storage st' ad)"
```

```
      proof (cases rule: transfer)
        case (Contract v t g adv c g' v' acc ct cn f st'')
        moreover from Contract have "adv = ad" using a1 by auto
        ultimately have "pref (ReadL_int (bal (accounts st ad)), storage st ad)" using ad_ev a2 by auto
        moreover have "ReadL_int (bal (accounts st ad)) = ReadL_int (bal (acc ad))" using
transfer_same[OF Contract(7)] `adv = ad` ad_ev by simp
        ultimately have "pref (ReadL_int (bal (acc ad)), storage st ad)" by simp
        moreover from a0 have "c = cname" using Contract(5) `adv = ad` by simp
        then have "ct = members" and "f = fb" using C1 Contract(6) by simp+
        moreover have "gas st ≥ costs (TRANSFER ad' ex) ev cd st" using 2 by (simp add:stmt.simps
split:if_split_asm)
        then have "gas (st⦇gas := gas st - costs (TRANSFER ad' ex) ev cd st⦈) < gas st" using
transfer_not_zero[of ad' ex ev cd st] by simp
        moreover have "g' ≤ gas (st⦇gas := gas st - costs (TRANSFER ad' ex) ev cd st⦈)" using
msel_ssel_expr_load_rexp_gas(3)[OF Contract(2)] msel_ssel_expr_load_rexp_gas(3)[OF Contract(3)] by
simp
        ultimately have "wpS (local.stmt fb (ffold (init members) (emptyEnv ad c (address ev) v')
(fmdom members)) emptyStore)
          (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⦈)" using assm[OF
all_gas_le[OF `gas (st⦇gas := gas st - costs (TRANSFER ad' ex) ev cd st⦈) < gas st` "1.IH"], THEN
conjunct2, THEN conjunct2, THEN conjunct1] ad_ev Contract `adv = ad` `c = cname` unfolding Qfi_def
by blast
        with `ct = members` `f=fb` have "gas st' ≤ gas (st⦇gas := g', accounts := acc, stack :=
emptyStore, memory := emptyStore⦈) ∧ postf (ReadL_int (bal (accounts st' ad)), storage st' ad)" un-
folding wpS_def wp_def using Contract(8,9) `adv = ad` by simp
        moreover from this have "gas st' ≤ gas st" using `g' ≤ gas (st⦇gas := gas st - costs
(TRANSFER ad' ex) ev cd st⦈)` by auto
        ultimately show ?thesis by simp
      next
        case (EOA v t g adv g' acc)
        then show ?thesis using a0 a1 by simp
      qed
    next
      fix e
      assume "local.stmt (TRANSFER ad' ex) ev cd st = Exception e"
      show "e = Gas ∨ e = Err" using Ex.nchotomy by simp
    qed
  qed
qed
```

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 -
  from wp_external_invoke_transfer have "Pfe ad iv st" using assms by blast
```

```
  then show ?thesis using Pfe_def by simp
qed

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 -
  from wp_external_invoke_transfer have "Pe ad iv st" using assms by blast
  then show ?thesis using Pe_def by simp
qed

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 -
  from wp_external_invoke_transfer have "Pi ad pre post st" using assms by blast
  then show ?thesis using Pi_def by simp
qed

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 -
  from wp_external_invoke_transfer have "Pfi ad pref postf st" using assms by blast
  then show ?thesis using Pfi_def by simp
qed

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)))"
  using assms(4-)
proof (induction rule:stmt.induct)
  case (1 ev cd st)
  show ?case
  proof elims
    fix st'
    assume *: "stmt SKIP ev cd st = Normal ((), st')"
       and "type (accounts st' ad) = Some (Contract cname)"
    then show "iv (storage st' ad) ⌈bal (accounts st' ad)⌉" using 1 skip[OF *] by simp
  qed
next
  case (2 lv ex ev cd st)
  show ?case
  proof elims
    fix st'
    assume *: "stmt (ASSIGN lv ex) ev cd st = Normal ((), st')"
       and "type (accounts st' ad) = Some (Contract cname)"
    then show "iv (storage st' ad) ⌈bal (accounts st' ad)⌉" using 2 by (cases rule: assign[OF *];simp)
  qed
next
  case (3 s1 s2 ev cd st)
  show ?case
  proof elims
    fix st'
    assume *: "stmt (COMP s1 s2) ev cd st = Normal ((), st')"
       and **: "type (accounts st' ad) = Some (Contract cname)"
      show "iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
    proof (cases rule: comp[OF *])
      case (1 st'')
      moreover from 3(4) have "type (accounts (st(|gas := gas st - costs (COMP s1 s2) ev cd st|)) ad)
= Some (Contract cname) ⟶ iv (storage (st(|gas := gas st - costs (COMP s1 s2) ev cd st|)) ad) ⌈bal
(accounts (st(|gas := gas st - costs (COMP s1 s2) ev cd st|)) ad)⌉" by auto
      ultimately have "type (accounts st'' ad) = Some (Contract cname) ⟶ iv (storage st'' ad) ⌈bal
(accounts st'' ad)⌉" using 3(1)[OF _ _ 3(3)] by fastforce
```

```
      then show ?thesis using 3(2)[OF _ _ _ 3(3)] 1 ** by fastforce
    qed
  qed
next
  case (4 ex s1 s2 ev cd st)
  show ?case
  proof elims
    fix st'
    assume a1: "local.stmt (ITE ex s1 s2) ev cd st = Normal ((), st')"
       and a2: "type (accounts st' ad) = Some (Contract cname)"
      show "iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
    proof (cases rule:ite[OF a1])
      case (1 g)
      have "∀st'. local.stmt s1 ev cd (st(|gas := g|)) = Normal ((), st') ∧
          type (accounts st' ad) = Some (Contract cname) ⟶
          iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
        apply (rule 4(1)) using 1 4(3) 4(4) by auto
      then show ?thesis using a2 1(3) by blast
    next
      case (2 g)
      have "∀st'. local.stmt s2 ev cd (st(|gas := g|)) = Normal ((), st') ∧
          type (accounts st' ad) = Some (Contract cname) ⟶
          iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
        apply (rule 4(2)) using 2 4(3) 4(4) true_neq_false[symmetric] by auto
      then show ?thesis using a2 2(3) by blast
    qed
  qed
next
  case (5 ex s0 ev cd st)
  show ?case
  proof elims
    fix st'
    assume a1: "local.stmt (WHILE ex s0) ev cd st = Normal ((), st')"
       and a2: "type (accounts st' ad) = Some (Contract cname)"
      show "iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
    proof (cases rule:while[OF a1])
      case (1 g st'')
      have "∀st'. local.stmt s0 ev cd (st(|gas := g|)) = Normal ((), st') ∧
          type (accounts st' ad) = Some (Contract cname) ⟶
          iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
        apply (rule 5(1)) using 1 5(3) 5(4) by auto
      then have *: "type (accounts st'' ad) = Some (Contract cname) ⟶
          iv (storage st'' ad) ⌈bal (accounts st'' ad)⌉" using 1(3) by simp
      have "∀st'. local.stmt (WHILE ex s0) ev cd st'' = Normal ((), st') ∧
          type (accounts st' ad) = Some (Contract cname) ⟶
          iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
        apply (rule 5(2)) using 1 5(3) 5(4) * by auto
      then show ?thesis using a2 1(4) by blast
    next
      case (2 g)
      then show ?thesis using a2 5(4) by simp
    qed
  qed
next
  case (6 i xe ev cd st)
  show ?case
  proof elims
    fix st'
    assume a1: "local.stmt (INVOKE i xe) ev cd st = Normal ((), st')"
       and a2: "type (accounts st' ad) = Some (Contract cname)"
    show "iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
    proof (cases rule:invoke[OF a1])
      case (1 ct fb fp f e_l cd_l k_l m_l g st'')
      from 6(2) have "ad ≠ address e_l" using msel_ssel_expr_load_rexp_gas(4)[OF 1(4)] ffold_init_ad
```

```
by simp
      have "∀st'. local.stmt f e_l cd_l (st⦇gas := g, stack := k_l, memory := m_l⦈) = Normal ((), st') ∧
type (accounts st' ad) = Some (Contract cname) ⟶
          iv (storage st' ad) ⌈bal (accounts st' ad)⌉" apply (rule 6(1)) using 1 6(3) ‘ad ≠ address e_l‘
by auto
      then show ?thesis using a2 1(5,6) by auto
    qed
  qed
next
  case (7 adex i xe val ev cd st)
  show ?case
  proof elims
    fix st'
    assume a1: "local.stmt (EXTERNAL adex i xe val) ev cd st = Normal ((), st')"
       and a2: "type (accounts st' ad) = Some (Contract cname)"
    show "iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
    proof (cases rule:external[OF a1])
      case (1 adv c g ct cn fb' v t g' v' fp f e_l cd_l k_l m_l g'' acc st'')
      then show ?thesis
      proof (cases "adv = ad")
        case True
        then have "type (acc ad) = Some (Contract c)" using transfer_type_same[OF 1(10)] 1(4) by auto
        moreover from ‘type (acc ad) = Some (Contract c)‘ have "type (accounts st' ad) = Some
(Contract c)" using atype_same[OF 1(11)] 1(12) by simp
        then have "c = cname" using a2 by simp
        moreover from ‘c = cname‘ have "ct = members" using 1 C1 by simp
        moreover have "g'' ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF 1(2)]
msel_ssel_expr_load_rexp_gas(3)[OF 1(6)] msel_ssel_expr_load_rexp_gas(4)[OF 1(9)] by linarith
        moreover have "iv (storage st ad) (ReadL_int (bal (acc ad)) - ReadL_int v')"
        proof -
          from ‘c = cname‘ have "type (accounts st ad) = Some (Contract cname)" using 1(4) True by
simp
          have "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" using 7(4) ‘type (accounts st ad)
= Some (Contract cname)‘ by simp
          moreover have "ReadL_int (bal (acc ad)) = ReadL_int (bal (accounts st ad)) + ReadL_int v'" using
transfer_add[OF 1(10)] 7(3) True by simp
          ultimately show ?thesis by simp
        qed
        ultimately have "wpS (local.stmt f e_l cd_l) (λst. iv (storage st ad) ⌈bal (accounts st ad)⌉)
(λe. e = Gas ∨ e = Err)
         (st⦇gas := g'', accounts := acc, stack := k_l, memory := m_l⦈)" using 1 True  using assms(1) 1(8)
7(3) unfolding Qe_def by simp
        then show ?thesis unfolding wpS_def wp_def using 1(11,12) by simp
      next
        case False
        then have *: "ad ≠ address e_l" using msel_ssel_expr_load_rexp_gas(4)[OF 1(9)] ffold_init_ad
by simp
        moreover have **:"type (acc ad) = Some (Contract cname) ⟶ iv (storage st ad) ⌈bal (acc ad)⌉"
        proof
          assume "type (acc ad) = Some (Contract cname)"
          then have "type (accounts st ad) = Some (Contract cname)" using transfer_type_same[OF 1(10)]
by simp
          then have "iv (storage st ad) ⌈bal (accounts st ad)⌉" using 7(4) by simp
          moreover have "bal (acc ad) = bal (accounts st ad)" using transfer_eq[OF 1(10)] 7(3) False
by simp
          ultimately show "iv (storage st ad) ⌈bal (acc ad)⌉" by simp
        qed
        ultimately have "∀st'. local.stmt f e_l cd_l (st⦇gas := g'', accounts := acc, stack := k_l, memory
:= m_l⦈) = Normal ((), st') ∧
         type (accounts st' ad) = Some (Contract cname) ⟶  iv (storage st' ad) ⌈bal (accounts st'
ad)⌉"
          using 7(1) 1 by auto
        then show ?thesis using a2 1(11,12) by simp
      qed
```

```
    next
      case (2 adv c g ct cn fb' v t g' v' acc st'')
      then show ?thesis
      proof (cases "adv = ad")
        case True
        then have "type (acc ad) = Some (Contract c)" using transfer_type_same[OF 2(9)] 2(4) by auto
        moreover from ‘type (acc ad) = Some (Contract c)‘ have "type (accounts st' ad) = Some
(Contract c)" using atype_same[OF 2(10)] 2(11) by simp
        then have "c = cname" using a2 by simp
        moreover from ‘c = cname‘ have "ct = members" and "fb'=fb" using 2 C1 by simp+
        moreover have "iv (storage st ad) (ReadL_int (bal (acc ad)) - ReadL_int v')"
        proof -
          from ‘c = cname‘ have "type (accounts st ad) = Some (Contract cname)" using 2(4) True by
simp
          then have "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" using 7(4) by simp
          moreover have "ReadL_int (bal (acc ad)) = ReadL_int (bal (accounts st ad)) + ReadL_int v'" using
transfer_add[OF 2(9)] 7(3) True by simp
          ultimately show "iv (storage st ad) (ReadL_int (bal (acc ad)) - ReadL_int v')" by simp
        qed
        moreover have "g' ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF 2(2)]
msel_ssel_expr_load_rexp_gas(3)[OF 2(6)] by linarith
        ultimately have "wpS (local.stmt fb' (ffold (init ct) (emptyEnv adv c (address ev) v') (fmdom
ct)) emptyStore) (λst. iv (storage st ad) ⌈bal (accounts st ad)⌉) (λe. e = Gas ∨ e = Err)
        (st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|))" using assms(2)
7(3) 2 True unfolding Qfe_def by simp
        then show ?thesis unfolding wpS_def wp_def using 2(10,11) by simp
      next
        case False
        moreover have **:"type (acc ad) = Some (Contract cname) ⟶ iv (storage st ad) ⌈bal (acc ad)⌉"
        proof
          assume "type (acc ad) = Some (Contract cname)"
          then have "type (accounts st ad) = Some (Contract cname)" using transfer_type_same[OF 2(9)]
by simp
          then have "iv (storage st ad) ⌈bal (accounts st ad)⌉" using 7(4) by simp
          moreover have "bal (acc ad) = bal (accounts st ad)" using transfer_eq[OF 2(9)] 7(3) False
by simp
          ultimately show "iv (storage st ad) ⌈bal (acc ad)⌉" by simp
        qed
        ultimately have "∀st'. local.stmt fb' (ffold (init ct) (emptyEnv adv c (address ev) v') (fmdom
ct)) emptyStore (st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|)) = Normal
((), st') ∧
         type (accounts st' ad) = Some (Contract cname) ⟶  iv (storage st' ad) ⌈bal (accounts st'
ad)⌉"
          using 7(2) 2 by auto
        then show ?thesis using a2 2(10,11) by simp
      qed
    qed
  qed
next
  case (8 ad' ex ev cd st)
  show ?case
  proof elims
    fix st'
    assume a1: "local.stmt (TRANSFER ad' ex) ev cd st = Normal ((), st')"
       and a2: "type (accounts st' ad) = Some (Contract cname)"
    show "iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
    proof (cases rule:transfer[OF a1])
      case (1 v t g adv c g' v' acc ct cn f st'')
      then show ?thesis
      proof (cases "adv = ad")
        case True
        then have "type (acc ad) = Some (Contract c)" using transfer_type_same[OF 1(7)] 1(5) by auto
        moreover from ‘type (acc ad) = Some (Contract c)‘ have "type (accounts st' ad) = Some
(Contract c)" using atype_same[OF 1(8)] 1(9) by simp
```

```
        then have "c = cname" using a2 by simp
        moreover from ‘c = cname‘ have "ct = members" and "f=fb" using 1 C1 by simp+
        moreover have "g' ≤ gas st" using msel_ssel_expr_load_rexp_gas(3)[OF 1(2)]
msel_ssel_expr_load_rexp_gas(3)[OF 1(3)] by linarith
        moreover have "iv (storage st ad) (ReadL_int (bal (acc ad)) - ReadL_int v')"
        proof -
          from ‘c = cname‘ have "type (accounts st ad) = Some (Contract cname)" using 1(5) True by
simp
          then have "iv (storage st ad) (ReadL_int (bal (accounts st ad)))" using 8(3) by simp
          moreover have "ReadL_int (bal (acc ad)) = ReadL_int (bal (accounts st ad)) + ReadL_int v'" using
transfer_add[OF 1(7)] 8(2) True by simp
          ultimately show "iv (storage st ad) (ReadL_int (bal (acc ad)) - ReadL_int v')" by simp
        qed
        ultimately have "wpS (local.stmt f (ffold (init ct) (emptyEnv adv c (address ev) v') (fmdom
ct)) emptyStore) (λst. iv (storage st ad) ⌈bal (accounts st ad)⌉) (λe. e = Gas ∨ e = Err)
         (st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|))" using assms(2)
8(2) 1 True unfolding Qfe_def by simp
        then show ?thesis unfolding wpS_def wp_def using 1(8,9) by simp
      next
        case False
        moreover have "type (acc ad) = Some (Contract cname) ⟶ iv (storage st ad) ⌈bal (acc ad)⌉"
        proof
          assume "type (acc ad) = Some (Contract cname)"
          then have "type (accounts st ad) = Some (Contract cname)" using transfer_type_same[OF 1(7)]
by simp
          then have "iv (storage st ad) ⌈bal (accounts st ad)⌉" using 8(3) by simp
          moreover have "bal (acc ad) = bal (accounts st ad)" using transfer_eq[OF 1(7)] 8(2) False
by simp
          ultimately show "iv (storage st ad) ⌈bal (acc ad)⌉" by simp
        qed
        ultimately have "∀st'. local.stmt f (ffold (init ct) (emptyEnv adv c (address ev) v') (fmdom
ct)) emptyStore
          (st(|gas := g', accounts := acc, stack := emptyStore, memory := emptyStore|)) = Normal ((),
st') ∧ type (accounts st' ad) = Some (Contract cname) ⟶
          iv (storage st' ad) ⌈bal (accounts st' ad)⌉" using 8(1) 1 by auto
        then show ?thesis using a2 1(8, 9) by simp
      qed
    next
      case (2 v t g adv g' v' acc)
      then show ?thesis
      proof (cases "adv = ad")
        case True
        then show ?thesis using 2(5,7) a2 transfer_type_same[OF 2(6)] by simp
      next
        case False
        then have "bal (acc ad) = bal (accounts st ad)" using transfer_eq[OF 2(6)] 8(2) by simp
        moreover have "type (accounts st ad) = Some (Contract cname)" using transfer_type_same[OF
2(6)] 2(7) a2 by simp
        then have "iv (storage st ad) ⌈bal (accounts st ad)⌉" using 8(3) by simp
        ultimately show ?thesis using 2(7) by simp
      qed
    qed
  qed
next
  case (9 id0 tp s e cd st)
  show ?case
  proof elims
    fix st'
    assume a1: "local.stmt (BLOCK ((id0, tp), None) s) e cd st = Normal ((), st')"
       and a2: "type (accounts st' ad) = Some (Contract cname)"
    show "iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
    proof (cases rule:blockNone[OF a1])
      case (1 cd' mem' sck' e')
      have "address e' = address e" using decl_env[OF 1(2)] by simp
```

```
      have "∀st'. local.stmt s e' cd' (st⦇gas := gas st - costs (BLOCK ((id0, tp), None) s) e cd st,
stack := sck',
           memory := mem'⦈) = Normal ((), st') ∧
          type (accounts st' ad) = Some (Contract cname) ⟶
          iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
        apply (rule 9(1)) using 1 9(2,3) ‘address e' = address e‘ by auto
      then show ?thesis using a2 1(3) by blast
    qed
  qed
next
  case (10 id0 tp ex' s e cd st)
  show ?case
  proof elims
    fix st'
    assume a1: "local.stmt (BLOCK ((id0, tp), Some ex') s) e cd st = Normal ((), st')"
       and a2: "type (accounts st' ad) = Some (Contract cname)"
    show "iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
    proof (cases rule:blockSome[OF a1])
      case (1 v t g cd' mem' sck' e')
      have "address e' = address e" using decl_env[OF 1(3)] by simp
      have "∀st'. local.stmt s e' cd' (st⦇gas := g, stack := sck', memory := mem'⦈) = Normal ((), st')
∧
          type (accounts st' ad) = Some (Contract cname) ⟶
          iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
        apply (rule 10(1)) using 1 10(2,3) ‘address e' = address e‘ by auto
      then show ?thesis using a2 1(4) by blast
    qed
  qed
next
  case (11 i xe val e cd st)
  show ?case
  proof elims
    fix st'
    assume a1: "local.stmt (NEW i xe val) e cd st = Normal ((), st')"
       and a2: "type (accounts st' ad) = Some (Contract cname)"
    show "iv (storage st' ad) ⌈bal (accounts st' ad)⌉"
    proof (cases rule:new[OF a1])
      case (1 v t g ct cn fb' e_l cd_l k_l m_l g' acc st'')
      define adv where "adv = hash (address e) ⌊contracts (accounts (st⦇gas := gas st - costs (NEW i xe
val) e cd st⦈) (address e))⌋"
      then have "address e_l = adv" using msel_ssel_expr_load_rexp_gas(4)[OF 1(5)] by simp
      then show ?thesis
      proof (cases "adv = ad")
        case True
        then show ?thesis
        proof (cases "i = cname")
          case t0: True
          then have "ct = members" and "cn = const" and "fb' = fb" using 1(4) C1 by simp+
          then have "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⦈)"
            using assms(3) 11(3) 1 True adv_def t0 unfolding constructor_def by auto
          then have "iv (storage st'' adv) ⌈bal (accounts st'' adv)⌉" unfolding wpS_def wp_def using
1(7) ‘cn = const‘ adv_def by simp
          then show ?thesis using 1(8) True by simp
        next
          case False
          moreover have "type (accounts st' ad) = Some (Contract i)"
          proof -
            from ‘adv = ad‘ have "type (((accounts st) (adv := ⦇bal = ShowL_int 0, type = Some (Contract
i), contracts = 0⦈)) ad) = Some (Contract i)" by simp
            then have "type (acc ad) = Some (Contract i)" using transfer_type_same[OF 1(6)] adv_def by
simp
```

```
          then have "type (accounts st'' ad) = Some (Contract i)" using atype_same[OF 1(7)] adv_def
by simp
          then show ?thesis using 1(8) by simp
        qed
        ultimately show ?thesis using a2 by simp
      qed
    next
      case False
      moreover have *: "type (acc ad) = Some (Contract cname) ⟶ iv (storage (st⦇storage:=(storage
st)(adv := {$$}), accounts:=acc⦈) ad) ⌈bal (acc ad)⌉"
      proof
        assume "type (acc ad) = Some (Contract cname)"
        then have "type (((accounts st) (adv := ⦇bal = ShowL_int 0, type = Some (Contract i),
contracts = 0⦈)) ad) = Some (Contract cname)" using transfer_type_same[OF 1(6)] adv_def by simp
        then have "type ((accounts st) ad) = Some (Contract cname)" using False by simp
        then have "iv (storage st ad) ⌈bal (accounts st ad)⌉" using 11(3) by simp
        then have "iv (storage (st⦇storage:=(storage st)(adv := {$$})⦈) ad) ⌈bal (accounts st ad)⌉"
using False by simp
        then show "iv (storage (st⦇storage:=(storage st)(adv := {$$}), accounts:=acc⦈) ad) ⌈bal (acc
ad)⌉" using transfer_eq[OF 1(6)] adv_def 11(2) False by simp
      qed
      ultimately have "∀st'. stmt (snd cn) e_l cd_l (st⦇gas := g', storage := (storage st) (adv :=
{$$}), accounts := acc, stack := k_l, memory := m_l⦈) = Normal ((), st') ∧
          type (accounts st' ad) = Some (Contract cname) ⟶ iv (storage st' ad) ⌈bal (accounts st'
ad)⌉"
        using 11(1)[where ?s'k4="st⦇gas := g', storage := (storage st) (adv := {$$}), accounts :=
acc, stack := k_l, memory := m_l⦈"]
          1 adv_def ‘address e_l = adv‘ False * by auto
      moreover have "type (accounts st'' ad) = Some (Contract cname)" using 1(8) a2 adv_def by auto
      ultimately show ?thesis using a2 1(6,7,8) adv_def by simp
    qed
  qed
qed

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)))"
  using assms invariant_rec by blast
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" by simp

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
  by unfold_locales auto

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()''"
  by eval
```

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))"
  using r1 unfolding bank_def by simp

lemma mdeposit[mcontract]:
  "members $$ STR ''deposit'' = Some (Method ([], True, deposit))"
  using r1 unfolding bank_def by simp
```

Variable lemma

```
lemma balance:
  "members $$ (STR ''balance'') = Some (Var (STMap TAddr (STValue (TUInt 256))))"
  using r1 bank_def fmlookup_of_list[of "[(STR ''balance'', Var (STMap TAddr (STValue (TUInt 256)))),
(STR ''deposit'', Method ([], True, ASSIGN myrexp (PLUS mylval VALUE))),
      (STR ''withdraw'', Method ([], True, BLOCK ((STR ''bal'', Value (TUInt 256)), Some mylval) (COMP
(ASSIGN myrexp (UINT 256 0)) Reentrancy.transfer)))]"] by simp
```

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 -
  from assms(1) r1 consider (withdraw) "mid = STR ''withdraw''" | (deposit) "mid = STR ''deposit''"
using bank_def fmap_of_list_SomeD[of "[(STR ''balance'', Var (STMap TAddr (STValue (TUInt 256)))), (STR
''deposit'', Method ([], True, deposit)), (STR ''withdraw'', Method ([], True, keep))]"] by auto
  then show ?thesis
  proof (cases)
    case withdraw
    then have "f = keep" and "fp = []" using assms(1) bank_def r1 fmlookup_of_list[of banklist] by
simp+
    then show ?thesis using assms(3) by simp
  next
    case deposit
    then have "f = deposit" and "fp = []" using assms(1) bank_def r1 fmlookup_of_list[of banklist] by
simp+
    then show ?thesis using assms(2) by simp
  qed
qed

lemma cases_int:
  assumes "members $$ mid = Some (Method (fp,False,f))"
    shows "P fp f"
  using assms r1 bank_def fmap_of_list_SomeD[of "[(STR ''balance'', Var (STMap TAddr (STValue (TUInt
256)))), (STR ''deposit'', Method ([], True, deposit)), (STR ''withdraw'', Method ([], True, keep))]"]
by auto

lemma cases_fb:
  assumes "P SKIP"
```

shows *"P fb"*
using *assms bank_def r2* **by** *simp*

lemma *cases_cons:*
assumes *"fst const = [] ⟹ P (fst const, SKIP)"*
shows *"P const"*
using *assms bank_def r3* **by** *simp*

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)))"*
using *assms* **unfolding** *iv_def* **by** *simp*

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)"*
using *assms(3)* **by** *(simp add:expr.simps createUInt_def split:if_split_asm)+*

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 -
have *"xe=[]"*
proof *(rule ccontr)*
assume *"¬ xe=[]"*
then obtain *x* **and** *xa* **where** *"xe=x # xa"* **by** *(meson list.exhaust)*
then have *"load True [] xe (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members)) emptyStore emptyStore emptyStore env cd (st⦇gas := g1⦈) g1 = throw Err g1"* **by** *(simp add:load.simps)*
with *assms* **show** *False* **by** *simp*
qed
then show *?thesis* **using** *assms(1)* **by** *simp*
qed

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 -
have *"fmlookup (denvalue (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members))) (STR ''balance'') = Some (Storage (STMap TAddr (STValue (TUInt 256))), Storeloc (STR ''balance''))"*

unfolding *emptyEnv_def* **using** *ffold_init_fmdom balance* **by** *simp*
moreover have *el_def: "e_l = (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members))"* **using** *load_empty_par[OF assms(1)]* **by** *(simp add:load.simps)*
ultimately have *"fmlookup (denvalue e_l) (STR ''balance'') = Some (Storage (STMap TAddr (STValue (TUInt 256))), Storeloc (STR ''balance''))"* **using** *assms* **by** *auto*
then have **: "fmlookup (denvalue e') (STR ''balance'') = Some (Storage (STMap TAddr (STValue (TUInt 256))), Storeloc (STR ''balance''))"* **using** *decl_env[OF assms(2)]* **by** *simp*

moreover obtain *g''* **where** *"ssel (STMap TAddr (STValue (TUInt 256))) (STR ''balance'') [SENDER] e' cd' (st0(|accounts := acc, stack := sck', memory := mem', gas := g'a|)) g'a = Normal (((address env) + (STR ''.'' + STR ''balance''), STValue (TUInt 256)), g'')"*
proof -
have *"g'a > $costs_e$ SENDER e' cd' (st0(|accounts := acc, stack := sck', memory := mem', gas := g'a|))"* **using** *assms(3) ** **by** *(simp add:expr.simps ssel.simps lexp.simps split:if_split_asm)*
then obtain *g''* **where** *"expr SENDER e' cd' (st0(|accounts := acc, stack := sck', memory := mem', gas := g'a|)) g'a = Normal ((KValue (sender e'), Value TAddr), g'')"* **by** *(simp add:expr.simps)*
moreover have *"sender e_l = address env"* **using** *el_def ffold_init_ad_same[of members "(emptyEnv ad cname (address env) v)" "fmdom members" e_l]* **unfolding** *emptyEnv_def* **by** *simp*
then have *"sender e' = address env"* **using** *decl_env[OF assms(2)]* **by** *simp*
ultimately show *?thesis* **using** *that hash_def* **by** *(simp add:ssel.simps)*
qed
ultimately show *"rv= LStoreloc (address env + (STR ''.'' + STR ''balance''))"* **and** *"rt=Storage (STValue (TUInt 256))"* **using** *assms(3)* **by** *(simp add:lexp.simps)+*
qed

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 -
from *assms(1)*
have *"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'' = rexp ((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'' - $costs_e$ (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''|)))"* **by** *(simp add:expr.simps split:if_split_asm)*
moreover have *"rexp ((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'' - $costs_e$ (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''|))) = Normal ((KValue (accessStorage (TUInt 256) (address env + (STR ''.'' + STR ''balance'')) s'), (Value (TUInt 256))),(g'' - $costs_e$ (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''|))))"*
proof -
from *assms(2)* **have** *"fmlookup (denvalue e') (STR ''bal'') = Some (Value (TUInt 256), Stackloc ⌊toploc k_l⌋)"* **by** *(simp add:push_def allocate_def updateStore_def)*
moreover have *"accessStore (⌊toploc k_l⌋) sck' = Some (KValue (accessStorage (TUInt 256) (address env + (STR ''.'' + STR ''balance'')) s'))"* **using** *assms(2)* **by** *(simp add:push_def allocate_def updateStore_def accessStore_def)*
ultimately show *?thesis* **by** *(simp add:rexp.simps)*
qed
ultimately have *"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 (accessStorage (TUInt 256) (address env + (STR ''.'' + STR ''balance'')) s'), (Value (TUInt 256))),(g'' - $costs_e$ (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''|))))"* **and** *"t = TUInt 256"* **using** *assms(1)* **by** *simp+*
then have *"lv = accessStorage (TUInt 256) (address env + (STR ''.'' + STR ''balance'')) s'"* **using** *assms(1)* **by** *(auto)*

```
  with ‘t = TUInt 256‘ show ?G1 and "t = TUInt 256" by simp+
qed

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 -
  have "fmlookup (denvalue (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members)))
(STR ''balance'') = Some (Storage (STMap TAddr (STValue (TUInt 256))), Storeloc (STR ''balance''))"
unfolding emptyEnv_def using ffold_init_fmdom balance by simp
  moreover have el_def: "e_l = (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom
members))" using load_empty_par[OF assms(1)] by (simp add:load.simps)
  ultimately have *: "fmlookup (denvalue e_l) (STR ''balance'') = Some (Storage (STMap TAddr (STValue
(TUInt 256))), Storeloc (STR ''balance''))" using assms by auto

  moreover obtain g'' where "ssel (STMap TAddr (STValue (TUInt 256))) (STR ''balance'') [SENDER] e_l
cd_l (st'(|gas := g2|)) g2 = Normal (((address env) + (STR ''.'' + STR ''balance''), STValue (TUInt 256)),
g'')"
  proof -
    have "g2 > costs_e SENDER e_l cd_l (st'(|gas := g2|))" using assms(2) * by (simp add:expr.simps
ssel.simps lexp.simps split:if_split_asm)
    then obtain g'' where "expr SENDER e_l cd_l (st'(|gas := g2|)) g2 = Normal ((KValue (sender e_l), Value
TAddr), g'')" by (simp add: expr.simps)
    moreover have "sender e_l = address env" using el_def ffold_init_ad_same[of members "(emptyEnv ad
cname (address env) v)" "fmdom members" e_l] unfolding emptyEnv_def by simp
    ultimately show ?thesis using that hash_def by (simp add:ssel.simps)
  qed
  ultimately show "rv= LStoreloc (address env + (STR ''.'' + STR ''balance''))" and "rt=Storage
(STValue (TUInt 256))" using assms(2) by (simp add: lexp.simps)+
qed

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 -
  have "fmlookup (denvalue (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members)))
(STR ''balance'') = Some (Storage (STMap TAddr (STValue (TUInt 256))), Storeloc (STR ''balance''))"
unfolding emptyEnv_def using ffold_init_fmdom balance by simp
  moreover have el_def: "e_l = (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom
members))" using load_empty_par[OF assms(1)] by (simp add:load.simps)
  ultimately have *: "fmlookup (denvalue e_l) (STR ''balance'') = Some (Storage (STMap TAddr (STValue
(TUInt 256))), Storeloc (STR ''balance''))" using assms by auto

  moreover obtain g'' where "ssel (STMap TAddr (STValue (TUInt 256))) (STR ''balance'') [SENDER] e_l
cd_l (st(|accounts := acc, stack := k_l, memory := m_l, gas := g2|)) g2 = Normal (((address env) + (STR ''.''
+ STR ''balance''), STValue (TUInt 256)), g'')"
  proof -
    have "g2 > costs_e SENDER e_l cd_l (st(|accounts := acc, stack := k_l, memory := m_l, gas := g2|))" using
assms(2) * by (simp add: expr.simps ssel.simps rexp.simps split:if_split_asm)
    then obtain g'' where "expr SENDER e_l cd_l (st(|accounts := acc, stack := k_l, memory := m_l, gas
:= g2|)) g2 = Normal ((KValue (sender e_l), Value TAddr), g'')" by (simp add:expr.simps ssel.simps
rexp.simps)
    moreover have "sender e_l = address env" using el_def ffold_init_ad_same[of members "(emptyEnv ad
cname (address env) v)" "fmdom members" e_l] unfolding emptyEnv_def by simp
    ultimately show ?thesis using that hash_def by (simp add:expr.simps ssel.simps rexp.simps)
  qed
```

```
  moreover have "ad = address e_l" using el_def ffold_init_ad_same[of members "(emptyEnv ad cname
(address env) v)" "fmdom members" e_l] unfolding emptyEnv_def by simp
  ultimately show "va= KValue (accessStorage (TUInt 256) (address env + (STR ''.'' + STR ''balance'')))
(storage st ad))" and "ta = Value (TUInt 256)" using assms(2) by (simp add:expr.simps ssel.simps
rexp.simps split:if_split_asm)+
qed

lemma balance_inj: "inj_on (λ(ad, x). (ad + (STR ''.'' + STR ''balance''),x)) {(ad, x). (fmlookup y ∘
f) ad = Some x}"
proof
  fix v v' assume asm1: "v ∈ {(ad, x). (fmlookup y ∘ f) ad = Some x}" and asm2: "v' ∈ {(ad, x).
(fmlookup y ∘ f) ad = Some x}"
  and *:"(case v of (ad, x) ⇒ (ad + (STR ''.'' + STR ''balance''),x)) = (case v' of (ad, x) ⇒ (ad +
(STR ''.'' + STR ''balance''),x))"
  then obtain ad ad' x x' where **: "v = (ad,x)" and ***: "v'=(ad',x')" by auto
  moreover from * ** *** have "ad + (STR ''.'' + STR ''balance'') = ad' + (STR ''.'' + STR
''balance'')" by simp
  with * ** have "ad = ad'" using String_Cancel[of ad "STR ''.'' + STR ''balance''" ad' ] by simp
  moreover from asm1 asm2 ** *** have "(fmlookup y ∘ f) ad = Some x" and "(fmlookup y ∘ f) ad' =
Some x'" by auto
  with calculation(3) have "x=x'" by simp
  ultimately show "v=v'" by simp
qed

lemma fmfinite: "finite ({(ad, x). fmlookup y ad = Some x})"
proof -
  have "{(ad, x). fmlookup y ad = Some x} ⊆ dom (fmlookup y) × ran (fmlookup y)"
  proof
    fix x assume "x ∈ {(ad, x). fmlookup y ad = Some x}"
    then have "fmlookup y (fst x) = Some (snd x)" by auto
    then have "fst x ∈ dom (fmlookup y)" and "snd x ∈ ran (fmlookup y)" using Map.ranI by
(blast,metis)
    then show "x ∈ dom (fmlookup y) × ran (fmlookup y)" by (simp add: mem_Times_iff)
  qed
  thus ?thesis by (simp add: finite_ran finite_subset)
qed

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 (cases rule: inj_on_finite[OF assms(1), of "{(ad, x)|ad x. (fmlookup y) ad = Some x}"])
  case 1
  then show ?case by auto
next
  case 2
  then have *: "finite {(ad, x) |ad x. fmlookup y ad = Some x}" using fmfinite[of y] by simp
  show ?case using finite_image_set[OF *, of "λ(ad, x). (ad, x)"] by simp
qed

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)"
  using assms by (auto simp add:expr.simps split:if_split_asm result.split_asm prod.split_asm
Stackvalue.split_asm Type.split_asm option.split_asm)

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 (cases "fmlookup s' (adr + (STR ''.'' + STR ''balance'')) = None")
```

```
  case True
  then have "accessStorage (TUInt 256) (adr + (STR ''.'' + STR ''balance'')) s' = ShowL_int 0" unfolding
accessStorage_def by simp
  moreover have "{(ad,x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x} = {(ad,x).
fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr}"
  proof
    show "{(ad, x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x} ⊆ {(ad, x). fmlookup s'
(ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr}"
    proof
      fix x
      assume "x ∈ {(ad, x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x}"
      then show "x ∈ {(ad, x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr}"
using True by auto
    qed
  next
    show "{(ad, x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr} ⊆ {(ad, x).
fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x }"
    proof
      fix x
      assume "x ∈ {(ad, x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr}"
      then show "x ∈ {(ad, x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x}" using True
by auto
    qed
  qed
  then have "SUMM s' = (∑ (ad,x)|fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠
adr. ReadL_int x)" by simp
  ultimately show ?thesis using Read_ShowL_id by simp
next
  case False
  then obtain val where val_def: "fmlookup s' (adr + (STR ''.'' + STR ''balance'')) = Some val" by
auto

  have "inj_on (λ(ad, x). (ad + (STR ''.'' + STR ''balance''), x)) {(ad, x). (fmlookup s' ∘ (λad. ad +
(STR ''.'' + STR ''balance'')))  ad = Some x}" using balance_inj by simp
  then have "finite {(ad, x). (fmlookup s' ∘ (λad. ad + (STR ''.'' + STR ''balance'')))  ad = Some x}"
using fmlookup_finite[of "λad. (ad + (STR ''.'' + STR ''balance''))" s'] by simp
  then have sum1: "finite ({(ad,x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠
adr})" using finite_subset[of "{(ad, x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad
≠ adr}" "{(ad, x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x}"] by auto
  moreover have sum2: "(adr,val) ∉ {(ad,x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x
∧ ad ≠ adr}" by simp
  moreover from sum1 val_def have "insert (adr,val) {(ad, x). fmlookup s' (ad + (STR ''.'' + STR
''balance'')) = Some x ∧ ad ≠ adr} = {(ad, x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some
x}" by auto
  ultimately show ?thesis using sum.insert[OF sum1 sum2, of "λ(ad,x). ReadL_int x"] val_def unfolding
accessStorage_def by simp
qed

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 -
  have "{(ad,x). fmlookup s'' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr} = {(ad,x).
fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr}"
  proof
    show "{(ad, x). fmlookup s'' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr}
       ⊆ {(ad, x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr}"
    proof
      fix xx
      assume "xx ∈ {(ad, x). fmlookup s'' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr}"
      then obtain ad x where "xx = (ad,x)" and "fmlookup s'' (ad + (STR ''.'' + STR ''balance'')) =
```

```
Some x" and "ad ≠ adr" by auto
      then have "fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x" using String_Cancel[of ad
"(STR ''.'' + STR ''balance'')" "adr"] s''_def by (simp split:if_split_asm)
      with ‘ad ≠ adr‘ ‘xx = (ad,x)‘ show "xx ∈ {(ad, x). fmlookup s' (ad + (STR ''.'' + STR
''balance'')) = Some x ∧ ad ≠ adr}" by simp
    qed
  next
    show "{(ad, x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr}
       ⊆ {(ad, x). fmlookup s'' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr}"
    proof
      fix xx
      assume "xx ∈ {(ad, x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr}"
      then obtain ad x where "xx = (ad,x)" and "fmlookup s' (ad + (STR ''.'' + STR ''balance'')) =
Some x" and "ad ≠ adr" by auto
      then have "fmlookup s'' (ad + (STR ''.'' + STR ''balance'')) = Some x" using String_Cancel[of ad
"(STR ''.'' + STR ''balance'')" "adr"] s''_def by (auto split:if_split_asm)
      with ‘ad ≠ adr‘ ‘xx = (ad,x)‘ show "xx ∈ {(ad, x). fmlookup s'' (ad + (STR ''.'' + STR
''balance'')) = Some x ∧ ad ≠ adr}" by simp
    qed
  qed
  thus ?thesis by simp
qed

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)(1 $$:= v')) ⌈bal (acc (address e_l))⌉) ∧
          (ad ≠ address e_l ⟶ iv (storage st0 ad) (ReadL_int (bal (acc ad))))"
proof (rule conjI; rule impI)
  assume "ad = address e_l"
  define s' s'' where "s' = storage st0 (address e_l)" and "s'' = storage st0 (address e_l)(1 $$:= v')"
  then have "s'' = fmupd 1 v' s'" by simp
  moreover from lexp_myrexp[OF assms(2) assms(5)] have "1 = address env + (STR ''.'' + STR
''balance'')" and "t'=TUInt 256" by simp+
  ultimately have s''_s': "s'' = s' (address env + (STR ''.'' + STR ''balance'') $$:= v')" by simp

  have "fmlookup (denvalue (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members)))
(STR ''balance'') = Some (Storage (STMap TAddr (STValue (TUInt 256))), Storeloc (STR ''balance''))"
unfolding emptyEnv_def using ffold_init_fmdom balance by simp
  moreover have "e_l = (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members))"
using load_empty_par[OF assms(2)] by (simp add:load.simps)
  ultimately have "fmlookup (denvalue e_l) (STR ''balance'') = Some (Storage (STMap TAddr (STValue
(TUInt 256))), Storeloc (STR ''balance''))" by simp
  then have va_def: "va = (createUInt 256 ((ReadL_int (accessStorage (TUInt 256) ((sender e_l) + (STR
''.'' + STR ''balance'')) s') + ReadL_int (svalue e_l))))"
   and "ta = (TUInt 256)"
  using assms(6) Read_ShowL_id unfolding s'_def by (auto split:if_split_asm simp add: expr.simps
ssel.simps rexp.simps add_def olift.simps hash_def)

  have "svalue e_l = v" and "sender e_l = address env" and "address e_l = ad" using ffold_init_ad_same
msel_ssel_expr_load_rexp_gas(4)[OF assms(2)] unfolding emptyEnv_def by simp+
  then have a_frame: "iv s' (ReadL_int (bal (acc ad)) - ReadL_int v)" using assms(4) unfolding s'_def by
simp

  from assms(1) have  "ReadL_int (bal (a ad)) + ReadL_int v < 2^256" using transfer_val2[OF assms(3)] by
simp
```

```
  moreover from ‘address env ≠ ad‘ have "ReadL_int (bal (acc ad)) = ReadL_int (bal (a ad)) + ReadL_int
v" using transfer_add[OF assms(3)] by simp
  ultimately have a_bal: "ReadL_int (bal (acc ad)) < 2^256" by simp

  moreover have a_bounds:
      "ReadL_int (accessStorage (TUInt 256) (sender e_l + (STR ''.'' + STR ''balance'')) s')  + ⌈svalue e_l⌉
< 2 ^ 256 ∧
      ReadL_int (accessStorage (TUInt 256) (sender e_l + (STR ''.'' + STR ''balance'')) s')  + ⌈svalue e_l⌉
≥ 0"
  proof (cases "fmlookup s' (sender e_l + (STR ''.'' + STR ''balance'')) = None")
    case True
    hence "(accessStorage (TUInt 256) (sender e_l + (STR ''.'' + STR ''balance'')) s') = ShowL_int 0"
unfolding accessStorage_def by simp
    moreover have "(⌈svalue e_l⌉::int) < 2 ^ 256"
    proof -
      from a_frame have "⌈svalue e_l⌉ + SUMM s' ≤ ReadL_int (bal (acc ad))" unfolding iv_def using
‘svalue e_l = v‘ by simp
      moreover have "0 ≤ SUMM s'"
      using a_frame sum_nonneg[of "{(ad,x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x}"
"λ(ad,x). ReadL_int x"] unfolding iv_def by auto
      ultimately have "⌈svalue e_l⌉ ≤ ReadL_int (bal (acc ad))" by simp
      then show "(⌈svalue e_l⌉::int) < 2 ^ 256" using a_bal by simp
    qed
    moreover have "ReadL_int v ≥ 0" using transfer_val1[OF assms(3)] by simp
    with ‘svalue e_l = v‘ have "(⌈svalue e_l⌉::int) ≥ 0" by simp
    ultimately show ?thesis using Read_ShowL_id by simp
  next
    case False
    then obtain x where x_def: "fmlookup s'  (sender e_l + (STR ''.'' + STR ''balance'')) = Some x" by
auto
    moreover from a_frame have "⌈svalue e_l⌉ + SUMM s' ≤ ReadL_int (bal (acc ad))" unfolding iv_def
using ‘svalue e_l = v‘ by simp
    moreover have "(case (sender e_l, x) of (ad, x) ⇒ ⌈x⌉) ≤ (∑ (ad, x)∈{(ad, x). fmlookup s' (ad +
(STR ''.'' + STR ''balance'')) = Some x}. ReadL_int x)"
    proof (cases rule: member_le_sum[of "(sender e_l,x)" "{(ad,x). fmlookup s' (ad + (STR ''.'' + STR
''balance'')) = Some x}" "λ(ad,x). ReadL_int x"])
      case 1
      then show ?case using x_def by simp
    next
      case (2 x)
      with a_frame show ?case unfolding iv_def by auto
    next
      case 3
      have "inj_on (λ(ad, x). (ad + (STR ''.'' + STR ''balance''), x)) {(ad, x). (fmlookup s' ∘ (λad.
ad + (STR ''.'' + STR ''balance'')))) ad = Some x}" using balance_inj by simp
      then have "finite {(ad, x). (fmlookup s' ∘ (λad. ad + (STR ''.'' + STR ''balance'')))) ad = Some
x}" using fmlookup_finite[of "λad. (ad + (STR ''.'' + STR ''balance''))" s'] by simp
      then show ?case using finite_subset[of "{(ad, x). fmlookup s' (ad + (STR ''.'' + STR
''balance'')) = Some x}" "{(ad, x). fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x}"] by
auto
    qed
    then have "ReadL_int x ≤ SUMM s'" by simp
    ultimately have "⌈svalue e_l⌉ + ReadL_int x ≤ ReadL_int (bal (acc ad))" by simp
    then have "⌈svalue e_l⌉ + ReadL_int x ≤ ReadL_int (bal (acc ad))" by simp
    with a_bal have "⌈svalue e_l⌉ + ReadL_int x < 2^256" by simp
    then have "⌈svalue e_l⌉ ≤ ⌈bal (acc ad)⌉ - SUMM s'" and lck: "fmlookup s' (sender e_l + (STR ''.'' +
STR ''balance'')) = Some x"  and "ReadL_int x + ⌈svalue e_l⌉ < 2 ^ 256" using a_frame x_def ‘svalue e_l =
v‘ unfolding iv_def by auto
    moreover from lck have "(accessStorage (TUInt 256) (sender e_l + (STR ''.'' + STR ''balance''))
s') = x" unfolding accessStorage_def by simp
    moreover have "⌈svalue e_l⌉ + ReadL_int x ≥ 0"
    proof -
      have "ReadL_int v ≥ 0" using transfer_val1[OF assms(3)] by simp
      with ‘svalue e_l = v‘ have "(⌈svalue e_l⌉::int) ≥ 0" by simp
```

```
    moreover from a_frame have "ReadL_int x ≥ 0" unfolding iv_def using x_def by simp
    ultimately show ?thesis by simp
  qed
  ultimately show ?thesis using Read_ShowL_id by simp
 qed
 ultimately have "va = ShowL_int (ReadL_int (accessStorage (TUInt 256) (sender e_l + (STR ''.'' + STR
''balance'')) s') + ReadL_int (svalue e_l))" using createUInt_id[where ?v="ReadL_int (accessStorage (TUInt
256) (sender e_l + (STR ''.'' + STR ''balance'')) s') + ReadL_int (svalue e_l)"] va_def by simp
 then have v'_def: "v' = ShowL_int (ReadL_int (accessStorage (TUInt 256) (address env + (STR ''.'' + STR
''balance'')) s') + ReadL_int v)" using ‘sender e_l = address env‘ ‘svalue e_l = v‘ ‘t'=TUInt 256‘ ‘ta =
(TUInt 256)‘ using assms(7) by auto

 have "SUMM s'' ≤ ⌈bal (acc ad)⌉"
 proof -
  have "SUMM s' ≤ ReadL_int (bal (acc ad)) - ReadL_int v" using a_frame unfolding iv_def by simp
  moreover have "SUMM s'' = SUMM s' + ReadL_int v"
  proof -
   from summ_eq_sum have s1: "SUMM s' = (∑ (ad,x)|fmlookup s' (ad + (STR ''.'' + STR ''balance'')))
= Some x ∧ ad ≠ address env. ReadL_int x) + ReadL_int (accessStorage (TUInt 256) (address env + (STR
''.'' + STR ''balance'')) s')" by simp
   have s2: "SUMM s'' = (∑ (ad,x)|fmlookup s'' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧
ad ≠ address env. ReadL_int x) + ReadL_int (accessStorage (TUInt 256) (address env + (STR ''.'' + STR
''balance'')) s') + ReadL_int v"
   proof -
    have "inj_on (λ(ad, x). (ad + (STR ''.'' + STR ''balance''), x)) {(ad, x). (fmlookup s'' ∘
(λad. ad + (STR ''.'' + STR ''balance'')))  ad = Some x}" using balance_inj by simp
    then have "finite {(ad, x). (fmlookup s'' ∘ (λad. ad + (STR ''.'' + STR ''balance'')))  ad =
Some x}" using fmlookup_finite[of "λad. (ad + (STR ''.'' + STR ''balance''))" s''] by simp
    then have sum1: "finite ({(ad,x). fmlookup s'' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧
ad ≠ address env})" using finite_subset[of "{(ad, x). fmlookup s'' (ad + (STR ''.'' + STR ''balance''))
= Some x ∧ ad ≠ address env}" "{(ad, x). fmlookup s'' (ad + (STR ''.'' + STR ''balance'')) = Some x}"]
by auto
    moreover have sum2: "(address env, ShowL_int (ReadL_int (accessStorage (TUInt 256) (address env
+ (STR ''.'' + STR ''balance'')) s') + ReadL_int v)) ∉ {(ad,x). fmlookup s'' (ad + (STR ''.'' + STR
''balance'')) = Some x ∧ ad ≠ address env}" by simp
    moreover from v'_def s''_s' have "insert (address env, ShowL_int (ReadL_int (accessStorage
(TUInt 256) (address env + (STR ''.'' + STR ''balance'')) s') + ReadL_int v)) {(ad, x). fmlookup s''
(ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ address env} = {(ad, x). fmlookup s'' (ad + (STR
''.'' + STR ''balance'')) = Some x}" by auto
    then show ?thesis using sum.insert[OF sum1 sum2, of "λ(ad,x). ReadL_int x"] Read_ShowL_id by
simp
   qed
   from sum_eq_update[OF s''_s'] have s3: "(∑ (ad,x)|fmlookup s'' (ad + (STR ''.'' + STR
''balance'')) = Some x ∧ ad ≠ address env. ReadL_int x)
                =(∑ (ad,x)|fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠
address env. ReadL_int x)" by simp
   moreover from s''_s' v'_def have "ReadL_int (accessStorage (TUInt 256) (address env + (STR
''.'' + STR ''balance'')) s'') = ReadL_int (accessStorage (TUInt 256) (address env + (STR ''.'' + STR
''balance'')) s') + ReadL_int v" using Read_ShowL_id unfolding accessStorage_def by simp
   ultimately have "SUMM s''= SUMM s' + ReadL_int v"
   proof -
    from s2 have "SUMM s'' = (∑ (ad,x)|fmlookup s'' (ad + (STR ''.'' + STR ''balance'')) = Some x
∧ ad ≠ address env. ReadL_int x) + ReadL_int (accessStorage (TUInt 256) (address env + (STR ''.'' + STR
''balance'')) s') + ReadL_int v" by simp
    also from s3 have "... = (∑ (ad,x)|fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x
∧ ad ≠ address env. ReadL_int x) + ReadL_int (accessStorage (TUInt 256) (address env + (STR ''.'' + STR
''balance'')) s') + ReadL_int v" by simp
    also from s1 have "... = SUMM s' - ReadL_int (accessStorage (TUInt 256) (address env + (STR
''.'' + STR ''balance'')) s') + ReadL_int (accessStorage (TUInt 256) (address env + (STR ''.'' + STR
''balance'')) s') + ReadL_int v" by simp
    finally show ?thesis by simp
   qed
   then show ?thesis by simp
  qed
```

```
    ultimately show ?thesis by simp
  qed
  moreover have "POS s''"
  proof (rule allI[OF allI])
    fix x xa
    show "fmlookup s'' (x + (STR ''.'' + STR ''balance'')) = Some xa ⟶ 0 ≤ (⌈xa⌉::int)"
    proof
      assume a_lookup: "fmlookup s'' (x + (STR ''.'' + STR ''balance'')) = Some xa"
      show "0 ≤ (⌈xa⌉::int)"
      proof (cases "x = address env")
        case True
        then show ?thesis using s''_s' a_lookup using Read_ShowL_id v'_def a_bounds ‘sender e_l =
address env‘ ‘svalue e_l = v‘ by auto
      next
        case False
        then have "fmlookup s' (x + (STR ''.'' + STR ''balance'')) = Some xa" using s''_s' a_lookup
String_Cancel[of "address env" "(STR ''.'' + STR ''balance'')" x] by (simp split:if_split_asm)
        then show ?thesis using a_frame unfolding iv_def by simp
      qed
    qed
  qed
  ultimately show "iv (storage st0 (address e_l)(1 $$:= v')) ⌈bal (acc (address e_l))⌉" unfolding iv_def
s''_def using ‘ad = address e_l‘ by simp
next
  assume "ad ≠ address e_l"
  moreover have "ad = address e_l" using ffold_init_ad_same msel_ssel_expr_load_rexp_gas(4)[OF
assms(2)] unfolding emptyEnv_def by simp
  ultimately show "iv (storage st0 ad) ⌈bal (acc ad)⌉" by simp
qed

lemma adapt_withdraw:
  fixes st acc sck' mem' g''a e' l v' xe
  defines "st' ≡ st(|accounts := acc, stack := sck', memory := mem', gas := g''a, storage := (storage
st) (address e' := (storage st (address e')) (1 $$:= v'))|)"
  assumes "iv (storage st ad) (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 -
  define acca where "acca = accounts st' ad"
  define s' where "s' = storage st (address e')"
  define s'a where "s'a = storage st' ad"
  moreover have "address e' = ad"
  proof -
```

```
    have "address e' = address e_l" using decl_env[OF assms(4)] by simp
    also have "address e_l = address (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom
members))" using msel_ssel_expr_load_rexp_gas(4)[OF assms(3)] by simp
    also have "... = ad" unfolding emptyEnv_def using ffold_init_ad_same by simp
    finally show ?thesis .
  qed
  ultimately have s'a_def: "s'a = fmupd 1 v' s'" unfolding s'_def st'_def by simp

  have "sender e' = address env"
  proof -
    have "sender e' = sender e_l" using decl_env[OF assms(4)] by simp
    also have "sender e_l = sender (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom
members))" using msel_ssel_expr_load_rexp_gas(4)[OF assms(3)] by simp
    also have "... = address env" unfolding emptyEnv_def using ffold_init_ad_same by simp
    finally show ?thesis .
  qed

  have "iv s'a (⌈bal acca⌉ - ⌈lv'⌉)" unfolding iv_def
  proof intros
    have "SUMM s' ≤ ⌈bal acca⌉"
    proof -
      from ‘address e' = ad‘ have "iv s' (ReadL_int (bal acca) - ReadL_int v)" using assms(2,5) unfold-
ing s'_def st'_def acca_def by simp
      then have "SUMM s' ≤ ⌈bal (acca)⌉ - ⌈v⌉" unfolding iv_def by simp
      moreover have "ReadL_int v ≥ 0" using transfer_val1[OF assms(9)] by simp
      ultimately show ?thesis by simp
    qed
    moreover have "SUMM s'a = SUMM s' - ReadL_int lv'"
    proof -
      from summ_eq_sum have "SUMM s'a = (∑ (ad,x)|fmlookup s'a (ad + (STR ''.'' + STR ''balance'')) 
= Some x ∧ ad ≠ sender e'. ReadL_int x) + ReadL_int (accessStorage (TUInt 256) (sender e' + (STR ''.'' +
STR ''balance'')) s'a)" by simp
      moreover from summ_eq_sum have "SUMM s' = (∑ (ad,x)|fmlookup s' (ad + (STR ''.'' + STR
''balance'')) = Some x ∧ ad ≠ sender e'. ReadL_int x) + ReadL_int (accessStorage (TUInt 256) (sender
e' + (STR ''.'' + STR ''balance'')) s')" by simp
      moreover from s'a_def lexp_myrexp_decl(1)[OF assms(3,4) assms(7)] have " s'a = s'((sender e' +
(STR ''.'' + STR ''balance''))$$:= v')" using ‘sender e' = address env‘ by simp
      with sum_eq_update have "(∑ (ad,x)|fmlookup s'a (ad + (STR ''.'' + STR ''balance'')) = Some x ∧
ad ≠ sender e'. ReadL_int x) = (∑ (ad,x)|fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad
≠ sender e'. ReadL_int x)"  by simp
      moreover have "ReadL_int (accessStorage (TUInt 256) (sender e' + (STR ''.'' + STR ''balance''))
s'a) = ReadL_int (accessStorage (TUInt 256) (sender e' + (STR ''.'' + STR ''balance'')) s') - ReadL_int
lv'"
      proof -
        have "ReadL_int (accessStorage (TUInt 256) (sender e' + (STR ''.'' + STR ''balance'')) s') =
ReadL_int lv'"
        proof -
          from expr_balance[OF assms(3) assms(8)] have "va= KValue (accessStorage (TUInt 256) (address
env + (STR ''.'' + STR ''balance'')) s')" and "ta = Value (TUInt 256)" using ‘address e' = ad‘ unfold-
ing s'_def by simp+
          then have "(sck',e') =  astack STR ''bal'' (Value (TUInt 256)) (KValue (accessStorage (TUInt
256) (address env + (STR ''.'' + STR ''balance'')) s')) (k_l, e_l)" using decl.simps(2) assms(4) by simp
          then have "ReadL_int (accessStorage (TUInt 256) (address env + (STR ''.'' + STR ''balance''))
s') = ReadL_int lv" and "t = TUInt 256" using expr_bal bal unfolding s'_def st'_def by auto
          moreover from ‘t = TUInt 256‘ have "lv = lv'" using con by simp
          ultimately show ?thesis using ‘sender e' = address env‘ by simp
        qed
        moreover have "ReadL_int (accessStorage (TUInt 256) (sender e' + (STR ''.'' + STR ''balance''))
s'a) = ReadL_int (ShowL_int 0)"
        proof -
          have "v' = ShowL_int 0"
          proof -
            have "vb = createUInt 256 0" and "tb=TUInt 256" using assms(5) by (simp add: expr.simps
load.simps split:if_split_asm)+
```

```
          moreover have "t'=TUInt 256" using lexp_myrexp_decl(2)[OF assms(3,4) assms(7)] by simp
          ultimately show ?thesis using assms(6) by (simp add: createUInt_id)
        qed
        moreover have "l= (sender e' + (STR ''.'' + STR ''balance''))" using lexp_myrexp_decl(1)[OF
assms(3,4) assms(7)] ‘sender e' = address env‘ by simp
        ultimately show ?thesis using s'a_def accessStorage_def by simp
      qed
      ultimately show ?thesis using Read_ShowL_id by simp
    qed
    ultimately show "SUMM s'a ≤ ⌈bal acca⌉ - ⌈lv'⌉" by simp
  next
    fix ad' x
    assume *: "fmlookup s'a (ad' + (STR ''.'' + STR ''balance'')) = Some x"
    show "0 ≤ ReadL_int x"
    proof (cases "ad' = sender e'")
      case True
      moreover have "v' = ShowL_int 0"
      proof -
        have "vb = createUInt 256 0" and "tb=TUInt 256" using assms(5) by (simp add:expr.simps
split:if_split_asm)+
        moreover have "t'=TUInt 256" using lexp_myrexp_decl(2)[OF assms(3,4) assms(7)] by simp
        ultimately show ?thesis using assms(6) by (simp add: createUInt_id)
      qed
      moreover have "l= (sender e' + (STR ''.'' + STR ''balance''))" using lexp_myrexp_decl(1)[OF
assms(3,4) assms(7)] ‘sender e' = address env‘ by simp
      ultimately show ?thesis using Read_ShowL_id s'a_def * by auto
    next
      case False
      moreover from ‘address e' = ad‘ have "iv s' (ReadL_int (bal (acc ad)) - ReadL_int v)" using
assms(2) unfolding s'_def by simp
      then have "POS s'" unfolding iv_def by simp
      moreover have "l= (sender e' + (STR ''.'' + STR ''balance''))" using lexp_myrexp_decl(1)[OF
assms(3,4) assms(7)] ‘sender e' = address env‘ by simp
      ultimately show ?thesis using s'a_def * String_Cancel by (auto split:if_split_asm)
    qed
  qed
  then show ?thesis unfolding s'a_def s'_def acca_def st'_def ‘address e' = ad‘ by simp
qed

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|))"
  apply (vcg_assign expr_rule: expr_plus[OF assms(5)] lexp_rule: lexp_myrexp(1)[OF assms(5)])
  by (simp add: adapt_deposit[OF assms(1,5,6,7)])

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 (rule meta_eq_to_obj_eq[THEN iffD1, OF Pfe_def assms(2),rule_format], (rule conjI; (rule
conjI)?))
  show "address e' = ad"
  proof -
    have "address e' = address e_l" using decl_env[OF assms(12)] by simp
    also have "address e_l = address (ffold (init members) (emptyEnv ad cname (address ev) gv') (fmdom
members))" using msel_ssel_expr_load_rexp_gas(4)[OF assms(9)] by simp
    also have "... = ad" unfolding emptyEnv_def using ffold_init_ad_same by simp
    finally show ?thesis .
  qed
next
  show "∀adv g. expr SENDER e' cd' (st'(|gas := gas st' - costs Reentrancy.transfer e' cd' st'|)) (gas
st' - costs Reentrancy.transfer e' cd' st') = Normal ((KValue adv, Value TAddr), g)
    ⟶ adv ≠ ad"
  proof (intros)
    fix adv g
    assume "expr SENDER e' cd' (st'(|gas := gas st' - costs Reentrancy.transfer e' cd' st'|)) (gas st' -
costs Reentrancy.transfer e' cd' st') = Normal ((KValue adv, Value TAddr), g)"
    moreover have "sender e' ≠ ad"
    proof -
      have "sender e' = sender e_l" using decl_env[OF assms(12)] by simp
      also have "sender e_l = sender (ffold (init members) (emptyEnv ad cname (address ev) gv') (fmdom
members))" using msel_ssel_expr_load_rexp_gas(4)[OF assms(9)] by simp
      also have "... = address ev" unfolding emptyEnv_def using ffold_init_ad_same by simp
      finally show ?thesis using assms(3) by simp
    qed
    ultimately show "adv ≠ ad" by (simp add:expr.simps split:result.split_asm if_split_asm
prod.split_asm)
  qed
next
  show "∀adv g v t g' v'.
       local.expr SENDER e' cd' (st'(|gas := gas st' - costs Reentrancy.transfer e' cd' st'|))
        (gas st' - costs Reentrancy.transfer e' cd' st') = Normal ((KValue adv, Value TAddr), g) ∧
       adv ≠ ad ∧
```

```
      local.expr (LVAL (L.Id STR ''bal'')) e' cd' (st'(|gas := g|)) g = Normal ((KValue v, Value t), g')
∧
      Valuetypes.convert t (TUInt 256) v = Some v' ⟶
      iv (storage st' ad) (⌈bal (accounts st' ad)⌉ - ReadL_int v')"
  proof elims
    fix adv lg lv lt lg' lv'
  assume a1:"expr SENDER e' cd' (st'(|gas := gas st' - costs Reentrancy.transfer e' cd' st'|)) (gas st' -
costs Reentrancy.transfer e' cd' st') =
         Normal ((KValue adv, Value TAddr), lg)"
    and adv_def: "adv ≠ ad"
    and bal: "expr (LVAL (L.Id STR ''bal'')) e' cd' (st'(|gas := lg|)) lg = Normal ((KValue lv, Value
lt), lg')"
    and con: "Valuetypes.convert lt (TUInt 256) lv = Some lv'"
   show "iv (storage st' ad) (⌈bal (accounts st' ad)⌉ - ReadL_int lv')"
    using adapt_withdraw[where ?acc=acc, OF assms(11) load_empty_par[OF assms(9)]
assms(12,16,13,14,15,10) _ adv_def _] a1 bal con unfolding st'_def by simp
  qed
qed

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|))"
  apply vcg_block_some
  apply vcg_comp
  apply (vcg_assign expr_rule: expr_0[OF assms(8)] lexp_rule: lexp_myrexp_decl[OF assms(8)])
  apply (rule wptransfer[OF _ assms(2-10)])
  apply (rule_tac assms(1)[THEN conjunct2,THEN conjunct2,THEN conjunct2])
    using assms(3,4) by simp

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|))"
  apply vcg_skip
  using weaken[where ?acc=acca, OF assms(2) transfer_val1[OF assms(1)]] by auto

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 -
  define adv where "adv = (hash (address ev) ⌊contracts (accounts st (address ev))⌋)"
  moreover have "ReadL_int (bal (((accounts st) (adv := (|bal = ShowL_int 0, type = Some (Contract i),
contracts = 0|))) adv)) = 0" using Read_ShowL_id[of 0] by simp
```

ultimately have *"ReadL$_{int}$ (bal (acc (hash (address ev) ⌊contracts (accounts st (address ev))⌋))) ≥ 0"*
using *transfer_mono[OF assms(1)]* **by** *simp*
then show *?thesis* **unfolding** *iv_def* **by** *simp*
qed

lemma *wp_true[external]:*
assumes *"E Gas"*
and *"E Err"*
shows *"wpS (stmt f e cd) (λst. True) E st"*
unfolding *wpS_def wp_def*
proof *((split result.split, (split prod.split)?); rule conjI; (rule allI)+; (rule impI)+)*
fix *x1 x1a s*
assume *"x1 = (x1a, s)"* **and** *"stmt f e cd st = Normal x1"*
then show *"gas s ≤ gas st ∧ True"* **using** *stmt_gas* **by** *simp*
next
fix *x2*
assume *"stmt f e cd st = Exception x2"*
show *"E x2"* **using** *assms Ex.nchotomy* **by** *metis*
qed

Final results

interpretation *vcg:VCG costs$_e$ ep costs cname members const fb "λ_. True" "λ_. True" "λ_ _. True" "λ_ _. True"*
by *(simp add: VCG.intro Calculus_axioms)*

lemma *safe_external: "Qe ad iv (st::State)"*
apply *(external cases: cases_ext)*
apply *(rule wp_deposit;assumption)*
apply *vcg_block_some*
apply *vcg_comp*
apply *(vcg_assign expr_rule: expr_0 lexp_rule: lexp_myrexp_decl)*
apply *(vcg.vcg_transfer_ext ad fallback_int: wp_true fallback_ext: wp_fallback cases_ext:cases_ext cases_int:cases_fb cases_fb:cases_int invariant:adapt_withdraw)*
done

lemma *safe_fallback: "Qfe ad iv st"*
apply *(fallback cases: cases_fb)*
apply *(rule wp_fallback; assumption)*
done

lemma *safe_constructor: "constructor iv"*
apply *(constructor cases: cases_cons)*
apply *vcg_skip*
apply *(simp add:wp_construct)*
done

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)))"
apply *(rule invariant)* **using** *assms safe_external safe_fallback safe_constructor* **by** *auto*

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

lemma update_bounds_int:
  assumes "eupdate ex = (E.INT b v)" and "b∈vbits"
  shows "(v < 2^(b-1)) ∧ v ≥ -(2^(b-1))"
proof (cases ex)
  case (INT b' v')
  then show ?thesis
  proof cases
    assume "b'∈vbits"
    show ?thesis
    proof cases
```

```
      let ?x="-(2^(b'-1)) + (v'+2^(b'-1)) mod 2^b'"
      assume "v'≥0"
      with ‘b'∈vbits‘ have "eupdate (E.INT b' v') = E.INT b' ?x" by simp
      with assms have "b=b'" and "v=?x" using INT by (simp,simp)
      moreover from ‘b'∈vbits‘ have "b'>0" by auto
      hence "?x < 2 ^(b'-1)" using upper_bound2[of b' "(v' + 2 ^ (b' - 1)) mod 2^b'"] by simp
      moreover have "?x ≥ -(2^(b'-1))" by simp
      ultimately show ?thesis by simp
    next
      let ?x="2^(b'-1) - (-v'+2^(b'-1)-1) mod (2^b') - 1"
      assume "¬v'≥0"
      with ‘b'∈vbits‘ have "eupdate (E.INT b' v') = E.INT b' ?x" by simp
      with assms have "b=b'" and "v=?x" using INT by (simp,simp)
      moreover have "(-v'+2^(b'-1)-1) mod (2^b')≥0" by simp
      hence "?x < 2 ^(b'-1)" by arith
      moreover from ‘b'∈vbits‘ have "b'>0" by auto
      hence "?x ≥ -(2^(b'-1))" using lower_bound2[of b' v'] by simp
      ultimately show ?thesis by simp
    qed
  next
    assume "¬ b'∈vbits"
    with assms show ?thesis using INT by simp
  qed
next
  case (UINT b' v')
  with assms show ?thesis
  proof cases
    assume "b'∈vbits"
    with assms show ?thesis using UINT by simp
  next
    assume "¬ b'∈vbits"
    with assms show ?thesis using UINT by simp
  qed
next
  case (ADDRESS x3)
  with assms show ?thesis by simp
next
  case (BALANCE x4)
  with assms show ?thesis by simp
next
  case THIS
  with assms show ?thesis by simp
next
  case SENDER
  with assms show ?thesis by simp
next
  case VALUE
  with assms show ?thesis by simp
next
  case TRUE
  with assms show ?thesis by simp
next
  case FALSE
  with assms show ?thesis by simp
next
  case (LVAL x7)
  with assms show ?thesis by simp
next
  case p: (PLUS e1 e2)
  show ?thesis
  proof (cases "eupdate e1")
    case i: (INT b1 v1)
    show ?thesis
    proof cases
```

```
      assume "b1∈vbits"
      show ?thesis
      proof (cases "eupdate e2")
        case i2: (INT b2 v2)
        then show ?thesis
        proof cases
          let ?v="v1+v2"
          assume "b2∈vbits"
          show ?thesis
          proof cases
            let ?x="-(2^((max b1 b2)-1)) + (?v+2^((max b1 b2)-1)) mod 2^(max b1 b2)"
            assume "?v≥0"
            with ‘b1∈vbits‘ ‘b2∈vbits‘ i i2 have "eupdate (PLUS e1 e2) = E.INT (max b1 b2) ?x" by
simp
            with assms have "b=max b1 b2" and "v=?x" using p by (simp,simp)
            moreover from ‘b1∈vbits‘ have "max b1 b2>0" by auto
            hence "?x < 2 ^(max b1 b2 - 1)"
              using upper_bound2[of "max b1 b2" "(?v + 2 ^ (max b1 b2 - 1)) mod 2^max b1 b2"] by simp
            moreover have "?x ≥ -(2^(max b1 b2-1))" by simp
            ultimately show ?thesis by simp
          next
            let ?x="2^((max b1 b2)-1) - (-?v+2^((max b1 b2)-1)-1) mod (2^(max b1 b2)) - 1"
            assume "¬?v≥0"
            with ‘b1∈vbits‘ ‘b2∈vbits‘ i i2 have "eupdate (PLUS e1 e2) = E.INT (max b1 b2) ?x" by
simp
            with assms have "b=max b1 b2" and "v=?x" using p by (simp,simp)
            moreover have "(-?v+2^(max b1 b2-1)-1) mod (2^max b1 b2)≥0" by simp
            hence "?x < 2 ^(max b1 b2-1)" by arith
            moreover from ‘b1∈vbits‘ have "max b1 b2>0" by auto
            hence "?x ≥ -(2^(max b1 b2-1))" using lower_bound2[of "max b1 b2" ?v] by simp
            ultimately show ?thesis by simp
          qed
        next
          assume "b2∉vbits"
          with p i i2 ‘b1∈vbits‘ show ?thesis using assms by simp
        qed
      next
        case u: (UINT b2 v2)
        then show ?thesis
        proof cases
          let ?v="v1+v2"
          assume "b2∈vbits"
          show ?thesis
          proof cases
            assume "b2<b1"
            then show ?thesis
            proof cases
              let ?x="(-(2^(b1-1)) + (?v+2^(b1-1)) mod (2^b1))"
              assume "?v≥0"
              with ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ i u have "eupdate (PLUS e1 e2) = E.INT b1 ?x" by
simp
              with assms have "b=b1" and "v=?x" using p by (simp,simp)
              moreover from ‘b1∈vbits‘ have "b1>0" by auto
              hence "?x < 2 ^(b1 - 1)" using upper_bound2[of b1] by simp
              moreover have "?x ≥ -(2^(b1-1))" by simp
              ultimately show ?thesis by simp
            next
              let ?x="2^(b1-1) - (-?v+2^(b1-1)-1) mod (2^b1) - 1"
              assume "¬?v≥0"
              with ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ i u have "eupdate (PLUS e1 e2) = E.INT b1 ?x" by
simp
              with assms have "b=b1" and "v=?x" using p i u by (simp,simp)
              moreover have "(-?v+2^(b1-1)-1) mod 2^b1≥0" by simp
              hence "?x < 2 ^(b1-1)" by arith
```

```
        moreover from ‘b1∈vbits‘ have "b1>0" by auto
        hence "?x ≥ -(2^(b1-1))" using lower_bound2[of b1 ?v] by simp
        ultimately show ?thesis by simp
      qed
    next
      assume "¬ b2<b1"
      with p i u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "b2∉vbits"
    with p i u ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (BALANCE x4)
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case THIS
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case SENDER
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case VALUE
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case TRUE
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case FALSE
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LVAL x7)
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (PLUS x81 x82)
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (MINUS x91 x92)
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (EQUAL x101 x102)
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LESS x111 x112)
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (AND x121 x122)
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (OR x131 x132)
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (NOT x131)
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (CALL x181 x182)
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (ECALL x191 x192 x193)
  with p i ‘b1∈vbits‘ show ?thesis using assms by simp
next
```

```
        case CONTRACTS
        with p i ‘b1∈vbits‘ show ?thesis using assms by simp
      qed
    next
      assume "¬ b1∈vbits"
      with p i show ?thesis using assms by simp
    qed
  next
    case u: (UINT b1 v1)
    show ?thesis
    proof cases
      assume "b1∈vbits"
      show ?thesis
      proof (cases "eupdate e2")
        case i: (INT b2 v2)
        then show ?thesis
        proof cases
          let ?v="v1+v2"
          assume "b2∈vbits"
          show ?thesis
          proof cases
            assume "b1<b2"
            then show ?thesis
            proof cases
              let ?x="(-(2^(b2-1)) + (?v+2^(b2-1)) mod (2^b2))"
              assume "?v≥0"
              with ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b1<b2‘ i u have "eupdate (PLUS e1 e2) = E.INT b2 ?x" by
simp
              with assms have "b=b2" and "v=?x" using p by (simp,simp)
              moreover from ‘b2∈vbits‘ have "b2>0" by auto
              hence "?x < 2 ^(b2 - 1)" using upper_bound2[of b2] by simp
              moreover have "?x ≥ -(2^(b2-1))" by simp
              ultimately show ?thesis by simp
            next
              let ?x="2^(b2-1) - (-?v+2^(b2-1)-1) mod (2^b2) - 1"
              assume "¬?v≥0"
              with ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b1<b2‘ i u have "eupdate (PLUS e1 e2) = E.INT b2 ?x" by
simp
              with assms have "b=b2" and "v=?x" using p i u by (simp,simp)
              moreover have "(-?v+2^(b2-1)-1) mod 2^b2≥0" by simp
              hence "?x < 2 ^(b2-1)" by arith
              moreover from ‘b2∈vbits‘ have "b2>0" by auto
              hence "?x ≥ -(2^(b2-1))" using lower_bound2[of b2 ?v] by simp
              ultimately show ?thesis by simp
            qed
          next
            assume "¬ b1<b2"
            with p i u ‘b1∈vbits‘ show ?thesis using assms by simp
          qed
        next
          assume "b2∉vbits"
          with p i u ‘b1∈vbits‘ show ?thesis using assms by simp
        qed
      next
        case u2: (UINT b2 v2)
        then show ?thesis
        proof cases
          assume "b2∈vbits"
          with ‘b1∈vbits‘ u u2 p show ?thesis using assms by simp
        next
          assume "¬b2∈vbits"
          with p u u2 ‘b1∈vbits‘ show ?thesis using assms by simp
        qed
      next
```

```
      case (ADDRESS x3)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (BALANCE x4)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case THIS
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case SENDER
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case VALUE
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case TRUE
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case FALSE
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (LVAL x7)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (PLUS x81 x82)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (MINUS x91 x92)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (EQUAL x101 x102)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (LESS x111 x112)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (AND x121 x122)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (OR x131 x132)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (NOT x131)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case CONTRACTS
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "¬ b1∈vbits"
    with p u show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with p show ?thesis using assms by simp
next
  case (BALANCE x4)
  with p show ?thesis using assms by simp
```

```
  next
    case THIS
    with p show ?thesis using assms by simp
  next
    case SENDER
    with p show ?thesis using assms by simp
  next
    case VALUE
    with p show ?thesis using assms by simp
  next
    case TRUE
    with p show ?thesis using assms by simp
  next
    case FALSE
    with p show ?thesis using assms by simp
  next
    case (LVAL x7)
    with p show ?thesis using assms by simp
  next
    case (PLUS x81 x82)
    with p show ?thesis using assms by simp
  next
    case (MINUS x91 x92)
    with p show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with p show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with p show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with p show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with p show ?thesis using assms by simp
  next
    case (NOT x131)
    with p show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with p show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with p show ?thesis using assms by simp
  next
    case CONTRACTS
    with p show ?thesis using assms by simp
  qed
next
  case m: (MINUS e1 e2)
  show ?thesis
  proof (cases "eupdate e1")
    case i: (INT b1 v1)
    with m show ?thesis
    proof cases
      assume "b1∈vbits"
      show ?thesis
      proof (cases "eupdate e2")
        case i2: (INT b2 v2)
        then show ?thesis
        proof cases
          let ?v="v1-v2"
          assume "b2∈vbits"
```

```
      with ‘b1 ∈ vbits‘ have
        u_def: "eupdate (MINUS e1 e2) =
        (let v = v1 - v2
          in if 0 ≤ v
            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))"
        using i i2 eupdate.simps(11)[of e1 e2] by simp
      show ?thesis
      proof cases
        let ?x="-(2^((max b1 b2)-1)) + (?v+2^((max b1 b2)-1)) mod 2^(max b1 b2)"
        assume "?v≥0"
        with u_def have "eupdate (MINUS e1 e2) = E.INT (max b1 b2) ?x" by simp
        with assms have "b=max b1 b2" and "v=?x" using m by (simp,simp)
        moreover from ‘b1∈vbits‘ have "max b1 b2>0" by auto
        hence "?x < 2 ^(max b1 b2 - 1)"
          using upper_bound2[of "max b1 b2" "(?v + 2 ^ (max b1 b2 - 1)) mod 2^max b1 b2"] by simp
        moreover have "?x ≥ -(2^(max b1 b2-1))" by simp
        ultimately show ?thesis by simp
      next
        let ?x="2^((max b1 b2)-1) - (-?v+2^((max b1 b2)-1)-1) mod (2^(max b1 b2)) - 1"
        assume "¬?v≥0"
        with u_def have "eupdate (MINUS e1 e2) = E.INT (max b1 b2) ?x" using u_def by simp
        with assms have "b=max b1 b2" and "v=?x" using m by (simp,simp)
        moreover have "(-?v+2^(max b1 b2-1)-1) mod (2^max b1 b2)≥0" by simp
        hence "?x < 2 ^(max b1 b2-1)" by arith
        moreover from ‘b1∈vbits‘ have "max b1 b2>0" by auto
        hence "?x ≥ -(2^(max b1 b2-1))" using lower_bound2[of "max b1 b2" ?v] by simp
        ultimately show ?thesis by simp
      qed
    next
      assume "b2∉vbits"
      with m i i2 ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    case u: (UINT b2 v2)
    then show ?thesis
    proof cases
      let ?v="v1-v2"
      assume "b2∈vbits"
      show ?thesis
      proof cases
        assume "b2<b1"
        with ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ have
          u_def: "eupdate (MINUS e1 e2) =
          (let v = v1 - v2
            in if 0 ≤ v
              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))"
          using i u eupdate.simps(11)[of e1 e2] by simp
        then show ?thesis
        proof cases
          let ?x="(-(2^(b1-1)) + (?v+2^(b1-1)) mod (2^b1))"
          assume "?v≥0"
          with u_def have "eupdate (MINUS e1 e2) = E.INT b1 ?x" by simp
          with assms have "b=b1" and "v=?x" using m by (simp,simp)
          moreover from ‘b1∈vbits‘ have "b1>0" by auto
          hence "?x < 2 ^(b1 - 1)" using upper_bound2[of b1] by simp
          moreover have "?x ≥ -(2^(b1-1))" by simp
          ultimately show ?thesis by simp
        next
          let ?x="2^(b1-1) - (-?v+2^(b1-1)-1) mod (2^b1) - 1"
          assume "¬?v≥0"
```

```
        with u_def have "eupdate (MINUS e1 e2) = E.INT b1 ?x" by simp
        with assms have "b=b1" and "v=?x" using m i u by (simp,simp)
        moreover have "(-?v+2^(b1-1)-1) mod 2^b1≥0" by simp
        hence "?x < 2 ^(b1-1)" by arith
        moreover from ‘b1∈vbits‘ have "b1>0" by auto
        hence "?x ≥ -(2^(b1-1))" using lower_bound2[of b1 ?v] by simp
        ultimately show ?thesis by simp
      qed
    next
      assume "¬ b2<b1"
      with m i u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "b2∉vbits"
    with m i u ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (BALANCE x4)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case THIS
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case SENDER
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case VALUE
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case TRUE
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case FALSE
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LVAL x7)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (PLUS x81 x82)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (MINUS x91 x92)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (EQUAL x101 x102)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LESS x111 x112)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (AND x121 x122)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (OR x131 x132)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (NOT x131)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (CALL x181 x182)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
```

```
    next
      case (ECALL x191 x192 x193)
      with m i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case CONTRACTS
      with m i ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "¬ b1∈vbits"
    with m i show ?thesis using assms by simp
  qed
next
  case u: (UINT b1 v1)
  show ?thesis
  proof cases
    assume "b1∈vbits"
    show ?thesis
    proof (cases "eupdate e2")
      case i: (INT b2 v2)
      then show ?thesis
      proof cases
        let ?v="v1-v2"
        assume "b2∈vbits"
        show ?thesis
        proof cases
          assume "b1<b2"
          with ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ have
            u_def: "eupdate (MINUS e1 e2) =
            (let v = v1 - v2
              in if 0 ≤ v
                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))"
            using i u eupdate.simps(11)[of e1 e2] by simp
          then show ?thesis
          proof cases
            let ?x="(-(2^(b2-1)) + (?v+2^(b2-1)) mod (2^b2))"
            assume "?v≥0"
            with u_def have "eupdate (MINUS e1 e2) = E.INT b2 ?x" by simp
            with assms have "b=b2" and "v=?x" using m by (simp,simp)
            moreover from ‘b2∈vbits‘ have "b2>0" by auto
            hence "?x < 2 ^(b2 - 1)" using upper_bound2[of b2] by simp
            moreover have "?x ≥ -(2^(b2-1))" by simp
            ultimately show ?thesis by simp
          next
            let ?x="2^(b2-1) - (-?v+2^(b2-1)-1) mod (2^b2) - 1"
            assume "¬?v≥0"
            with u_def have "eupdate (MINUS e1 e2) = E.INT b2 ?x" by simp
            with assms have "b=b2" and "v=?x" using m i u by (simp,simp)
            moreover have "(-?v+2^(b2-1)-1) mod 2^b2≥0" by simp
            hence "?x < 2 ^(b2-1)" by arith
            moreover from ‘b2∈vbits‘ have "b2>0" by auto
            hence "?x ≥ -(2^(b2-1))" using lower_bound2[of b2 ?v] by simp
            ultimately show ?thesis by simp
          qed
        next
          assume "¬ b1<b2"
          with m i u ‘b1∈vbits‘ show ?thesis using assms by simp
        qed
      next
        assume "b2∉vbits"
        with m i u ‘b1∈vbits‘ show ?thesis using assms by simp
      qed
    next
      case u2: (UINT b2 v2)
```

```
    then show ?thesis
    proof cases
      assume "b2∈vbits"
      with ‘b1∈vbits‘ u u2 m show ?thesis using assms by simp
    next
      assume "¬b2∈vbits"
      with m u u2 ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    case (ADDRESS x3)
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case (BALANCE x4)
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case THIS
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case SENDER
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case VALUE
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case TRUE
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case FALSE
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case (LVAL x7)
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case (PLUS x81 x82)
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case (MINUS x91 x92)
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case (NOT x131)
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case CONTRACTS
    with m u ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
```

```
      assume "¬ b1∈vbits"
      with m u show ?thesis using assms by simp
    qed
  next
    case (ADDRESS x3)
    with m show ?thesis using assms by simp
  next
    case (BALANCE x4)
    with m show ?thesis using assms by simp
  next
    case THIS
    with m show ?thesis using assms by simp
  next
    case SENDER
    with m show ?thesis using assms by simp
  next
    case VALUE
    with m show ?thesis using assms by simp
  next
    case TRUE
    with m show ?thesis using assms by simp
  next
    case FALSE
    with m show ?thesis using assms by simp
  next
    case (LVAL x7)
    with m show ?thesis using assms by simp
  next
    case (PLUS x81 x82)
    with m show ?thesis using assms by simp
  next
    case (MINUS x91 x92)
    with m show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with m show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with m show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with m show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with m show ?thesis using assms by simp
  next
    case (NOT x131)
    with m show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with m show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with m show ?thesis using assms by simp
  next
    case CONTRACTS
    with m show ?thesis using assms by simp
  qed
next
  case e: (EQUAL e1 e2)
  show ?thesis
  proof (cases "eupdate e1")
    case i: (INT b1 v1)
    show ?thesis
```

```
proof cases
  assume "b1∈vbits"
  show ?thesis
  proof (cases "eupdate e2")
    case i2: (INT b2 v2)
    then show ?thesis
    proof cases
      assume "b2∈vbits"
      show ?thesis
      proof cases
        assume "v1=v2"
        with assms show ?thesis using e i i2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
      next
        assume "¬ v1=v2"
        with assms show ?thesis using e i i2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
      qed
    next
      assume "b2∉vbits"
      with e i i2 ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    case u: (UINT b2 v2)
    then show ?thesis
    proof cases
      assume "b2∈vbits"
      show ?thesis
      proof cases
        assume "b2<b1"
        then show ?thesis
        proof cases
          assume "v1=v2"
          with assms show ?thesis using e i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ by simp
        next
          assume "¬ v1=v2"
          with assms show ?thesis using e i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ by simp
        qed
      next
        assume "¬ b2<b1"
        with e i u ‘b1∈vbits‘ show ?thesis using assms by simp
      qed
    next
      assume "b2∉vbits"
      with e i u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    case (ADDRESS x3)
    with e i ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case (BALANCE x4)
    with e i ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case THIS
    with e i ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case SENDER
    with e i ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case VALUE
    with e i ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case TRUE
    with e i ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case FALSE
```

```
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (LVAL x7)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (PLUS x81 x82)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (MINUS x91 x92)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (EQUAL x101 x102)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (LESS x111 x112)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (AND x121 x122)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (OR x131 x132)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (NOT x131)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case CONTRACTS
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "¬ b1∈vbits"
    with e i show ?thesis using assms by simp
  qed
next
  case u: (UINT b1 v1)
  show ?thesis
  proof cases
    assume "b1∈vbits"
    show ?thesis
    proof (cases "eupdate e2")
      case i: (INT b2 v2)
      then show ?thesis
      proof cases
        assume "b2∈vbits"
        show ?thesis
        proof cases
          assume "b1<b2"
          then show ?thesis
          proof cases
            assume "v1=v2"
            with assms show ?thesis using e i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b1<b2‘ by simp
          next
            assume "¬ v1=v2"
            with assms show ?thesis using e i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b1<b2‘ by simp
          qed
        next
          assume "¬ b1<b2"
          with e i u ‘b1∈vbits‘ show ?thesis using assms by simp
```

```
    qed
  next
    assume "b2∉vbits"
    with e i u ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
  case u2: (UINT b2 v2)
  then show ?thesis
  proof cases
    assume "b2∈vbits"
    show ?thesis
    proof cases
      assume "v1=v2"
      with assms show ?thesis using e u u2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
    next
      assume "¬ v1=v2"
      with assms show ?thesis using e u u2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
    qed
  next
    assume "¬b2∈vbits"
    with e u u2 ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (BALANCE x4)
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case THIS
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case SENDER
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case VALUE
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case TRUE
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case FALSE
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LVAL x7)
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (PLUS x81 x82)
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (MINUS x91 x92)
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (EQUAL x101 x102)
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LESS x111 x112)
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (AND x121 x122)
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (OR x131 x132)
  with e u ‘b1∈vbits‘ show ?thesis using assms by simp
```

```
    next
      case (NOT x131)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case CONTRACTS
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "¬ b1∈vbits"
    with e u show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with e show ?thesis using assms by simp
next
  case (BALANCE x4)
  with e show ?thesis using assms by simp
next
  case THIS
  with e show ?thesis using assms by simp
next
  case SENDER
  with e show ?thesis using assms by simp
next
  case VALUE
  with e show ?thesis using assms by simp
next
  case TRUE
  with e show ?thesis using assms by simp
next
  case FALSE
  with e show ?thesis using assms by simp
next
  case (LVAL x7)
  with e show ?thesis using assms by simp
next
  case (PLUS x81 x82)
  with e show ?thesis using assms by simp
next
  case (MINUS x91 x92)
  with e show ?thesis using assms by simp
next
  case (EQUAL x101 x102)
  with e show ?thesis using assms by simp
next
  case (LESS x111 x112)
  with e show ?thesis using assms by simp
next
  case (AND x121 x122)
  with e show ?thesis using assms by simp
next
  case (OR x131 x132)
  with e show ?thesis using assms by simp
next
  case (NOT x131)
  with e show ?thesis using assms by simp
next
  case (CALL x181 x182)
```

```
    with e show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with e show ?thesis using assms by simp
  next
    case CONTRACTS
    with e show ?thesis using assms by simp
  qed
next
  case l: (LESS e1 e2)
  show ?thesis
  proof (cases "eupdate e1")
    case i: (INT b1 v1)
    show ?thesis
    proof cases
      assume "b1∈vbits"
      show ?thesis
      proof (cases "eupdate e2")
        case i2: (INT b2 v2)
        then show ?thesis
        proof cases
          assume "b2∈vbits"
          show ?thesis
          proof cases
            assume "v1<v2"
            with assms show ?thesis using l i i2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
          next
            assume "¬ v1<v2"
            with assms show ?thesis using l i i2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
          qed
        next
          assume "b2∉vbits"
          with l i i2 ‘b1∈vbits‘ show ?thesis using assms by simp
        qed
      next
        case u: (UINT b2 v2)
        then show ?thesis
        proof cases
          assume "b2∈vbits"
          show ?thesis
          proof cases
            assume "b2<b1"
            then show ?thesis
            proof cases
              assume "v1<v2"
              with assms show ?thesis using l i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ by simp
            next
              assume "¬ v1<v2"
              with assms show ?thesis using l i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ by simp
            qed
          next
            assume "¬ b2<b1"
            with l i u ‘b1∈vbits‘ show ?thesis using assms by simp
          qed
        next
          assume "b2∉vbits"
          with l i u ‘b1∈vbits‘ show ?thesis using assms by simp
        qed
      next
        case (ADDRESS x3)
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case (BALANCE x4)
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
```

```
      next
        case THIS
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case SENDER
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case VALUE
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case TRUE
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case FALSE
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case (LVAL x7)
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case (PLUS x81 x82)
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case (MINUS x91 x92)
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case (EQUAL x101 x102)
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case (LESS x111 x112)
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case (AND x121 x122)
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case (OR x131 x132)
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case (NOT x131)
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case (CALL x181 x182)
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case (ECALL x191 x192 x193)
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      next
        case CONTRACTS
        with l i ‘b1∈vbits‘ show ?thesis using assms by simp
      qed
    next
      assume "¬ b1∈vbits"
      with l i show ?thesis using assms by simp
    qed
  next
    case u: (UINT b1 v1)
    show ?thesis
    proof cases
      assume "b1∈vbits"
      show ?thesis
      proof (cases "eupdate e2")
        case i: (INT b2 v2)
        then show ?thesis
        proof cases
          assume "b2∈vbits"
```

```
    show ?thesis
    proof cases
      assume "b1<b2"
      then show ?thesis
      proof cases
        assume "v1<v2"
        with assms show ?thesis using l i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b1<b2‘ by simp
      next
        assume "¬ v1<v2"
        with assms show ?thesis using l i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b1<b2‘ by simp
      qed
    next
      assume "¬ b1<b2"
      with l i u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "b2∉vbits"
    with l i u ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
  case u2: (UINT b2 v2)
  then show ?thesis
  proof cases
    assume "b2∈vbits"
    show ?thesis
    proof cases
      assume "v1<v2"
      with assms show ?thesis using l u u2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
    next
      assume "¬ v1<v2"
      with assms show ?thesis using l u u2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
    qed
  next
    assume "¬b2∈vbits"
    with l u u2 ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (BALANCE x4)
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case THIS
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case SENDER
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case VALUE
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case TRUE
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case FALSE
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LVAL x7)
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (PLUS x81 x82)
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
```

```
      case (MINUS x91 x92)
      with l u `b1∈vbits` show ?thesis using assms by simp
    next
      case (EQUAL x101 x102)
      with l u `b1∈vbits` show ?thesis using assms by simp
    next
      case (LESS x111 x112)
      with l u `b1∈vbits` show ?thesis using assms by simp
    next
      case (AND x121 x122)
      with l u `b1∈vbits` show ?thesis using assms by simp
    next
      case (OR x131 x132)
      with l u `b1∈vbits` show ?thesis using assms by simp
    next
      case (NOT x131)
      with l u `b1∈vbits` show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with l u `b1∈vbits` show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with l u `b1∈vbits` show ?thesis using assms by simp
    next
      case CONTRACTS
      with l u `b1∈vbits` show ?thesis using assms by simp
    qed
  next
    assume "¬ b1∈vbits"
    with l u show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with l show ?thesis using assms by simp
next
  case (BALANCE x4)
  with l show ?thesis using assms by simp
next
  case THIS
  with l show ?thesis using assms by simp
next
  case SENDER
  with l show ?thesis using assms by simp
next
  case VALUE
  with l show ?thesis using assms by simp
next
  case TRUE
  with l show ?thesis using assms by simp
next
  case FALSE
  with l show ?thesis using assms by simp
next
  case (LVAL x7)
  with l show ?thesis using assms by simp
next
  case (PLUS x81 x82)
  with l show ?thesis using assms by simp
next
  case (MINUS x91 x92)
  with l show ?thesis using assms by simp
next
  case (EQUAL x101 x102)
  with l show ?thesis using assms by simp
```

```
  next
    case (LESS x111 x112)
    with l show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with l show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with l show ?thesis using assms by simp
  next
    case (NOT x131)
    with l show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with l show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with l show ?thesis using assms by simp
  next
    case CONTRACTS
    with l show ?thesis using assms by simp
  qed
next
  case a: (AND e1 e2)
  show ?thesis
  proof (cases "eupdate e1")
    case (INT x11 x12)
    with a show ?thesis using assms by simp
  next
    case (UINT x21 x22)
    with a show ?thesis using assms by simp
  next
    case (ADDRESS x3)
    with a show ?thesis using assms by simp
  next
    case (BALANCE x4)
    with a show ?thesis using assms by simp
  next
    case THIS
    with a show ?thesis using assms by simp
  next
    case SENDER
    with a show ?thesis using assms by simp
  next
    case VALUE
    with a show ?thesis using assms by simp
  next
    case t: TRUE
    show ?thesis
    proof (cases "eupdate e2")
      case (INT x11 x12)
      with a t show ?thesis using assms by simp
    next
      case (UINT x21 x22)
      with a t show ?thesis using assms by simp
    next
      case (ADDRESS x3)
      with a t show ?thesis using assms by simp
    next
      case (BALANCE x4)
      with a t show ?thesis using assms by simp
    next
      case THIS
      with a t show ?thesis using assms by simp
```

```
    next
      case SENDER
      with a t show ?thesis using assms by simp
    next
      case VALUE
      with a t show ?thesis using assms by simp
    next
      case TRUE
      with a t show ?thesis using assms by simp
    next
      case FALSE
      with a t show ?thesis using assms by simp
    next
      case (LVAL x7)
      with a t show ?thesis using assms by simp
    next
      case (PLUS x81 x82)
      with a t show ?thesis using assms by simp
    next
      case (MINUS x91 x92)
      with a t show ?thesis using assms by simp
    next
      case (EQUAL x101 x102)
      with a t show ?thesis using assms by simp
    next
      case (LESS x111 x112)
      with a t show ?thesis using assms by simp
    next
      case (AND x121 x122)
      with a t show ?thesis using assms by simp
    next
      case (OR x131 x132)
      with a t show ?thesis using assms by simp
    next
      case (NOT x131)
      with a t show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with a t show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with a t show ?thesis using assms by simp
    next
      case CONTRACTS
      with a t show ?thesis using assms by simp
    qed
  next
    case f: FALSE
    show ?thesis
    proof (cases "eupdate e2")
      case (INT x11 x12)
      with a f show ?thesis using assms by simp
    next
      case (UINT x21 x22)
      with a f show ?thesis using assms by simp
    next
      case (ADDRESS x3)
      with a f show ?thesis using assms by simp
    next
      case (BALANCE x4)
      with a f show ?thesis using assms by simp
    next
      case THIS
      with a f show ?thesis using assms by simp
```

```
  next
    case SENDER
    with a f show ?thesis using assms by simp
  next
    case VALUE
    with a f show ?thesis using assms by simp
  next
    case TRUE
    with a f show ?thesis using assms by simp
  next
    case FALSE
    with a f show ?thesis using assms by simp
  next
    case (LVAL x7)
    with a f show ?thesis using assms by simp
  next
    case (PLUS x81 x82)
    with a f show ?thesis using assms by simp
  next
    case (MINUS x91 x92)
    with a f show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with a f show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with a f show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with a f show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with a f show ?thesis using assms by simp
  next
    case (NOT x131)
    with a f show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with a f show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with a f show ?thesis using assms by simp
  next
    case CONTRACTS
    with a f show ?thesis using assms by simp
  qed
next
  case (LVAL x7)
  with a show ?thesis using assms by simp
next
  case (PLUS x81 x82)
  with a show ?thesis using assms by simp
next
  case (MINUS x91 x92)
  with a show ?thesis using assms by simp
next
  case (EQUAL x101 x102)
  with a show ?thesis using assms by simp
next
  case (LESS x111 x112)
  with a show ?thesis using assms by simp
next
  case (AND x121 x122)
  with a show ?thesis using assms by simp
```

```
  next
    case (OR x131 x132)
    with a show ?thesis using assms by simp
  next
    case (NOT x131)
    with a show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with a show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with a show ?thesis using assms by simp
  next
    case CONTRACTS
    with a show ?thesis using assms by simp
  qed
next
  case o: (OR e1 e2)
  show ?thesis
  proof (cases "eupdate e1")
    case (INT x11 x12)
    with o show ?thesis using assms by simp
  next
    case (UINT x21 x22)
    with o show ?thesis using assms by simp
  next
    case (ADDRESS x3)
    with o show ?thesis using assms by simp
  next
    case (BALANCE x4)
    with o show ?thesis using assms by simp
  next
    case THIS
    with o show ?thesis using assms by simp
  next
    case SENDER
    with o show ?thesis using assms by simp
  next
    case VALUE
    with o show ?thesis using assms by simp
  next
    case t: TRUE
    show ?thesis
    proof (cases "eupdate e2")
      case (INT x11 x12)
      with o t show ?thesis using assms by simp
    next
      case (UINT x21 x22)
      with o t show ?thesis using assms by simp
    next
      case (ADDRESS x3)
      with o t show ?thesis using assms by simp
    next
      case (BALANCE x4)
      with o t show ?thesis using assms by simp
    next
      case THIS
      with o t show ?thesis using assms by simp
    next
      case SENDER
      with o t show ?thesis using assms by simp
    next
      case VALUE
      with o t show ?thesis using assms by simp
```

```
  next
    case TRUE
    with o t show ?thesis using assms by simp
  next
    case FALSE
    with o t show ?thesis using assms by simp
  next
    case (LVAL x7)
    with o t show ?thesis using assms by simp
  next
    case (PLUS x81 x82)
    with o t show ?thesis using assms by simp
  next
    case (MINUS x91 x92)
    with o t show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with o t show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with o t show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with o t show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with o t show ?thesis using assms by simp
  next
    case (NOT x131)
    with o t show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with o t show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with o t show ?thesis using assms by simp
  next
    case CONTRACTS
    with o t show ?thesis using assms by simp
  qed
next
  case f: FALSE
  show ?thesis
  proof (cases "eupdate e2")
    case (INT x11 x12)
    with o f show ?thesis using assms by simp
  next
    case (UINT x21 x22)
    with o f show ?thesis using assms by simp
  next
    case (ADDRESS x3)
    with o f show ?thesis using assms by simp
  next
    case (BALANCE x4)
    with o f show ?thesis using assms by simp
  next
    case THIS
    with o f show ?thesis using assms by simp
  next
    case SENDER
    with o f show ?thesis using assms by simp
  next
    case VALUE
    with o f show ?thesis using assms by simp
```

```
    next
      case TRUE
      with o f show ?thesis using assms by simp
    next
      case FALSE
      with o f show ?thesis using assms by simp
    next
      case (LVAL x7)
      with o f show ?thesis using assms by simp
    next
      case (PLUS x81 x82)
      with o f show ?thesis using assms by simp
    next
      case (MINUS x91 x92)
      with o f show ?thesis using assms by simp
    next
      case (EQUAL x101 x102)
      with o f show ?thesis using assms by simp
    next
      case (LESS x111 x112)
      with o f show ?thesis using assms by simp
    next
      case (AND x121 x122)
      with o f show ?thesis using assms by simp
    next
      case (OR x131 x132)
      with o f show ?thesis using assms by simp
    next
      case (NOT x131)
      with o f show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with o f show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with o f show ?thesis using assms by simp
    next
      case CONTRACTS
      with o f show ?thesis using assms by simp
    qed
  next
    case (LVAL x7)
    with o show ?thesis using assms by simp
  next
    case (PLUS x81 x82)
    with o show ?thesis using assms by simp
  next
    case (MINUS x91 x92)
    with o show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with o show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with o show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with o show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with o show ?thesis using assms by simp
  next
    case (NOT x131)
    with o show ?thesis using assms by simp
```

```
  next
    case (CALL x181 x182)
    with o show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with o show ?thesis using assms by simp
  next
    case CONTRACTS
    with o show ?thesis using assms by simp
  qed
next
  case o: (NOT e1)
  show ?thesis
  proof (cases "eupdate e1")
    case (INT x11 x12)
    with o show ?thesis using assms by simp
  next
    case (UINT x21 x22)
    with o show ?thesis using assms by simp
  next
    case (ADDRESS x3)
    with o show ?thesis using assms by simp
  next
    case (BALANCE x4)
    with o show ?thesis using assms by simp
  next
    case THIS
    with o show ?thesis using assms by simp
  next
    case SENDER
    with o show ?thesis using assms by simp
  next
    case VALUE
    with o show ?thesis using assms by simp
  next
    case t: TRUE
    with o show ?thesis using assms by simp
  next
    case f: FALSE
    with o show ?thesis using assms by simp
  next
    case (LVAL x7)
    with o show ?thesis using assms by simp
  next
    case (PLUS x81 x82)
    with o show ?thesis using assms by simp
  next
    case (MINUS x91 x92)
    with o show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with o show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with o show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with o show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with o show ?thesis using assms by simp
  next
    case (NOT x131)
    with o show ?thesis using assms by simp
```

```
  next
    case (CALL x181 x182)
    with o show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with o show ?thesis using assms by simp
  next
    case CONTRACTS
    with o show ?thesis using assms by simp
  qed
next
  case (CALL x181 x182)
  with assms show ?thesis by simp
next
  case (ECALL x191 x192 x193)
  with assms show ?thesis by simp
next
  case CONTRACTS
  with assms show ?thesis by simp
qed

lemma update_bounds_uint:
  assumes "eupdate ex = UINT b v" and "b∈vbits"
  shows "v < 2^b ∧ v ≥ 0"
proof (cases ex)
  case (INT b' v')
  with assms show ?thesis
  proof cases
    assume "b'∈vbits"
    show ?thesis
    proof cases
      assume "v'≥0"
      with INT show ?thesis using assms ‘b'∈vbits‘ by simp
    next
      assume "¬ v'≥0"
      with INT show ?thesis using assms ‘b'∈vbits‘ by simp
    qed
  next
    assume "¬ b'∈vbits"
    with INT show ?thesis using assms by simp
  qed
next
  case (UINT b' v')
  then show ?thesis
  proof cases
    assume "b'∈vbits"
    with UINT show ?thesis using assms by auto
  next
    assume "¬ b'∈vbits"
    with UINT show ?thesis using assms by auto
  qed
next
  case (ADDRESS x3)
  with assms show ?thesis by simp
next
  case (BALANCE x4)
  with assms show ?thesis by simp
next
  case THIS
  with assms show ?thesis by simp
next
  case SENDER
  with assms show ?thesis by simp
next
```

```
  case VALUE
  with assms show ?thesis by simp
next
  case TRUE
  with assms show ?thesis by simp
next
  case FALSE
  with assms show ?thesis by simp
next
  case (LVAL x7)
  with assms show ?thesis by simp
next
  case p: (PLUS e1 e2)
  show ?thesis
  proof (cases "eupdate e1")
    case i: (INT b1 v1)
    with p show ?thesis
    proof cases
      assume "b1∈vbits"
      show ?thesis
      proof (cases "eupdate e2")
        case i2: (INT b2 v2)
        then show ?thesis
        proof cases
          let ?v="v1+v2"
          assume "b2∈vbits"
          show ?thesis
          proof cases
            assume "?v≥0"
            with assms show ?thesis using p i i2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
          next
            assume "¬?v≥0"
            with assms show ?thesis using p i i2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
          qed
        next
          assume "b2∉vbits"
          with p i i2 ‘b1∈vbits‘ show ?thesis using assms by simp
        qed
      next
        case u: (UINT b2 v2)
        then show ?thesis
        proof cases
          let ?v="v1+v2"
          assume "b2∈vbits"
          show ?thesis
          proof cases
            assume "b2<b1"
            then show ?thesis
            proof cases
              assume "?v≥0"
              with assms show ?thesis using p i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ by simp
            next
              assume "¬?v≥0"
              with assms show ?thesis using p i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ by simp
            qed
          next
            assume "¬ b2<b1"
            with p i u ‘b1∈vbits‘ show ?thesis using assms by simp
          qed
        next
          assume "b2∉vbits"
          with p i u ‘b1∈vbits‘ show ?thesis using assms by simp
        qed
      next
```

```
      case (ADDRESS x3)
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (BALANCE x4)
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case THIS
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case SENDER
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case VALUE
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case TRUE
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case FALSE
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (LVAL x7)
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (PLUS x81 x82)
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (MINUS x91 x92)
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (EQUAL x101 x102)
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (LESS x111 x112)
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (AND x121 x122)
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (OR x131 x132)
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (NOT x131)
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case CONTRACTS
      with p i ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "¬ b1∈vbits"
    with p i show ?thesis using assms by simp
  qed
next
  case u: (UINT b1 v1)
  with p show ?thesis
  proof cases
    assume "b1∈vbits"
    show ?thesis
```

```
proof (cases "eupdate e2")
  case i: (INT b2 v2)
  then show ?thesis
  proof cases
    let ?v="v1+v2"
    assume "b2∈vbits"
    show ?thesis
    proof cases
      assume "b1<b2"
      then show ?thesis
      proof cases
        assume "?v≥0"
        with assms show ?thesis using p i u ‘b1∈vbits‘  ‘b2∈vbits‘ ‘b1<b2‘ by simp
      next
        assume "¬?v≥0"
        with assms show ?thesis using p i u ‘b1∈vbits‘  ‘b2∈vbits‘ ‘b1<b2‘ by simp
      qed
    next
      assume "¬ b1<b2"
      with p i u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "b2∉vbits"
    with p i u ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
  case u2: (UINT b2 v2)
  then show ?thesis
  proof cases
    let ?x="((v1 + v2) mod (2^(max b1 b2)))"
    assume "b2∈vbits"
    with ‘b1∈vbits‘ u u2 have "eupdate (PLUS e1 e2) = UINT (max b1 b2) ?x" by simp
    with assms have "b=max b1 b2" and "v=?x" using p by (simp,simp)
    moreover from ‘b1∈vbits‘ have "max b1 b2>0" by auto
    hence "?x < 2 ^(max b1 b2)" by simp
    moreover have "?x ≥ 0" by simp
    ultimately show ?thesis by simp
  next
    assume "¬b2∈vbits"
    with p u u2 ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with p u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (BALANCE x4)
  with p u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case THIS
  with p u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case SENDER
  with p u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case VALUE
  with p u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case TRUE
  with p u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case FALSE
  with p u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LVAL x7)
```

```
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (PLUS x81 x82)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (MINUS x91 x92)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (EQUAL x101 x102)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (LESS x111 x112)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (AND x121 x122)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (OR x131 x132)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (NOT x131)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case CONTRACTS
      with p u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "¬ b1∈vbits"
    with p u show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with p show ?thesis using assms by simp
next
  case (BALANCE x4)
  with p show ?thesis using assms by simp
next
  case THIS
  with p show ?thesis using assms by simp
next
  case SENDER
  with p show ?thesis using assms by simp
next
  case VALUE
  with p show ?thesis using assms by simp
next
  case TRUE
  with p show ?thesis using assms by simp
next
  case FALSE
  with p show ?thesis using assms by simp
next
  case (LVAL x7)
  with p show ?thesis using assms by simp
next
  case (PLUS x81 x82)
  with p show ?thesis using assms by simp
next
```

```
    case (MINUS x91 x92)
    with p show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with p show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with p show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with p show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with p show ?thesis using assms by simp
  next
    case (NOT x131)
    with p show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with p show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with p show ?thesis using assms by simp
  next
    case CONTRACTS
    with p show ?thesis using assms by simp
  qed
next
  case m: (MINUS e1 e2)
  show ?thesis
  proof (cases "eupdate e1")
    case i: (INT b1 v1)
    with m show ?thesis
    proof cases
      assume "b1∈vbits"
      show ?thesis
      proof (cases "eupdate e2")
        case i2: (INT b2 v2)
        then show ?thesis
        proof cases
          let ?v="v1-v2"
          assume "b2∈vbits"
          show ?thesis
          proof cases
            assume "?v≥0"
            with assms show ?thesis using m i i2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
          next
            assume "¬?v≥0"
            with assms show ?thesis using m i i2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
          qed
        next
          assume "b2∉vbits"
          with m i i2 ‘b1∈vbits‘ show ?thesis using assms by simp
        qed
      next
        case u: (UINT b2 v2)
        then show ?thesis
        proof cases
          let ?v="v1-v2"
          assume "b2∈vbits"
          show ?thesis
          proof cases
            assume "b2<b1"
            show ?thesis
```

```
      proof cases
        assume "?v≥0"
        with assms show ?thesis using m i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ by simp
      next
        assume "¬?v≥0"
        with assms show ?thesis using m i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ by simp
      qed
    next
      assume "¬ b2<b1"
      with m i u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "b2∉vbits"
    with m i u ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (BALANCE x4)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case THIS
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case SENDER
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case VALUE
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case TRUE
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case FALSE
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LVAL x7)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (PLUS x81 x82)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (MINUS x91 x92)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (EQUAL x101 x102)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LESS x111 x112)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (AND x121 x122)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (OR x131 x132)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (NOT x131)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (CALL x181 x182)
  with m i ‘b1∈vbits‘ show ?thesis using assms by simp
next
```

```
      case (ECALL x191 x192 x193)
      with m i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case CONTRACTS
      with m i ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "¬ b1∈vbits"
    with m i show ?thesis using assms by simp
  qed
next
  case u: (UINT b1 v1)
  with m show ?thesis
  proof cases
    assume "b1∈vbits"
    show ?thesis
    proof (cases "eupdate e2")
      case i: (INT b2 v2)
      then show ?thesis
      proof cases
        let ?v="v1-v2"
        assume "b2∈vbits"
        show ?thesis
        proof cases
          assume "b1<b2"
          show ?thesis
          proof cases
            assume "?v≥0"
            with assms show ?thesis using m i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b1<b2‘ by simp
          next
            assume "¬?v≥0"
            with assms show ?thesis using m i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b1<b2‘ by simp
          qed
        next
          assume "¬ b1<b2"
          with m i u ‘b1∈vbits‘ show ?thesis using assms by simp
        qed
      next
        assume "b2∉vbits"
        with m i u ‘b1∈vbits‘ show ?thesis using assms by simp
      qed
    next
      case u2: (UINT b2 v2)
      then show ?thesis
      proof cases
        let ?x="((v1 - v2) mod (2^(max b1 b2)))"
        assume "b2∈vbits"
        with ‘b1∈vbits‘ u u2 have "eupdate (MINUS e1 e2) = UINT (max b1 b2) ?x" by simp
        with assms have "b=max b1 b2" and "v=?x" using m by (simp,simp)
        moreover from ‘b1∈vbits‘ have "max b1 b2>0" by auto
        hence "?x < 2 ^(max b1 b2)" by simp
        moreover have "?x ≥ 0" by simp
        ultimately show ?thesis by simp
      next
        assume "¬b2∈vbits"
        with m u u2 ‘b1∈vbits‘ show ?thesis using assms by simp
      qed
    next
      case (ADDRESS x3)
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (BALANCE x4)
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
```

```
      case THIS
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case SENDER
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case VALUE
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case TRUE
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case FALSE
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (LVAL x7)
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (PLUS x81 x82)
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (MINUS x91 x92)
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (EQUAL x101 x102)
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (LESS x111 x112)
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (AND x121 x122)
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (OR x131 x132)
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (NOT x131)
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case CONTRACTS
      with m u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "¬ b1∈vbits"
    with m u show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with m show ?thesis using assms by simp
next
  case (BALANCE x4)
  with m show ?thesis using assms by simp
next
  case THIS
  with m show ?thesis using assms by simp
next
  case SENDER
  with m show ?thesis using assms by simp
```

```
  next
    case VALUE
    with m show ?thesis using assms by simp
  next
    case TRUE
    with m show ?thesis using assms by simp
  next
    case FALSE
    with m show ?thesis using assms by simp
  next
    case (LVAL x7)
    with m show ?thesis using assms by simp
  next
    case (PLUS x81 x82)
    with m show ?thesis using assms by simp
  next
    case (MINUS x91 x92)
    with m show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with m show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with m show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with m show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with m show ?thesis using assms by simp
  next
    case (NOT x131)
    with m show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with m show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with m show ?thesis using assms by simp
  next
    case CONTRACTS
    with m show ?thesis using assms by simp
  qed
next
  case e: (EQUAL e1 e2)
  show ?thesis
  proof (cases "eupdate e1")
    case i: (INT b1 v1)
    show ?thesis
    proof cases
      assume "b1∈vbits"
      show ?thesis
      proof (cases "eupdate e2")
        case i2: (INT b2 v2)
        then show ?thesis
        proof cases
          assume "b2∈vbits"
          show ?thesis
          proof cases
            assume "v1=v2"
            with assms show ?thesis using e i i2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
          next
            assume "¬ v1=v2"
            with assms show ?thesis using e i i2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
```

```
    qed
  next
    assume "b2∉vbits"
    with e i i2 ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
  case u: (UINT b2 v2)
  then show ?thesis
  proof cases
    assume "b2∈vbits"
    show ?thesis
    proof cases
      assume "b2<b1"
      then show ?thesis
      proof cases
        assume "v1=v2"
        with assms show ?thesis using e i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ by simp
      next
        assume "¬ v1=v2"
        with assms show ?thesis using e i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ by simp
      qed
    next
      assume "¬ b2<b1"
      with e i u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "b2∉vbits"
    with e i u ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with e i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (BALANCE x4)
  with e i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case THIS
  with e i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case SENDER
  with e i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case VALUE
  with e i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case TRUE
  with e i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case FALSE
  with e i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LVAL x7)
  with e i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (PLUS x81 x82)
  with e i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (MINUS x91 x92)
  with e i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (EQUAL x101 x102)
  with e i ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LESS x111 x112)
```

```
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (AND x121 x122)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (OR x131 x132)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (NOT x131)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case CONTRACTS
      with e i ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "¬ b1∈vbits"
    with e i show ?thesis using assms by simp
  qed
next
  case u: (UINT b1 v1)
  show ?thesis
  proof cases
    assume "b1∈vbits"
    show ?thesis
    proof (cases "eupdate e2")
      case i: (INT b2 v2)
      then show ?thesis
      proof cases
        let ?v="v1+v2"
        assume "b2∈vbits"
        show ?thesis
        proof cases
          assume "b1<b2"
          then show ?thesis
          proof cases
            assume "v1=v2"
            with assms show ?thesis using e i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b1<b2‘ by simp
          next
            assume "¬ v1=v2"
            with assms show ?thesis using e i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b1<b2‘ by simp
          qed
        next
          assume "¬ b1<b2"
          with e i u ‘b1∈vbits‘ show ?thesis using assms by simp
        qed
      next
        assume "b2∉vbits"
        with e i u ‘b1∈vbits‘ show ?thesis using assms by simp
      qed
    next
      case u2: (UINT b2 v2)
      then show ?thesis
      proof cases
        assume "b2∈vbits"
        show ?thesis
        proof cases
          assume "v1=v2"
          with assms show ?thesis using e u u2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
```

```
        next
          assume "¬ v1=v2"
          with assms show ?thesis using e u u2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
        qed
      next
        assume "¬b2∈vbits"
        with e u u2 ‘b1∈vbits‘ show ?thesis using assms by simp
      qed
    next
      case (ADDRESS x3)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (BALANCE x4)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case THIS
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case SENDER
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case VALUE
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case TRUE
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case FALSE
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (LVAL x7)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (PLUS x81 x82)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (MINUS x91 x92)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (EQUAL x101 x102)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (LESS x111 x112)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (AND x121 x122)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (OR x131 x132)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (NOT x131)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case CONTRACTS
      with e u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
```

```
      assume "¬ b1∈vbits"
      with e u show ?thesis using assms by simp
    qed
  next
    case (ADDRESS x3)
    with e show ?thesis using assms by simp
  next
    case (BALANCE x4)
    with e show ?thesis using assms by simp
  next
    case THIS
    with e show ?thesis using assms by simp
  next
    case SENDER
    with e show ?thesis using assms by simp
  next
    case VALUE
    with e show ?thesis using assms by simp
  next
    case TRUE
    with e show ?thesis using assms by simp
  next
    case FALSE
    with e show ?thesis using assms by simp
  next
    case (LVAL x7)
    with e show ?thesis using assms by simp
  next
    case (PLUS x81 x82)
    with e show ?thesis using assms by simp
  next
    case (MINUS x91 x92)
    with e show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with e show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with e show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with e show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with e show ?thesis using assms by simp
  next
    case (NOT x131)
    with e show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with e show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with e show ?thesis using assms by simp
  next
    case CONTRACTS
    with e show ?thesis using assms by simp
  qed
next
  case l: (LESS e1 e2)
  show ?thesis
  proof (cases "eupdate e1")
    case i: (INT b1 v1)
    show ?thesis
```

```
proof cases
  assume "b1∈vbits"
  show ?thesis
  proof (cases "eupdate e2")
    case i2: (INT b2 v2)
    then show ?thesis
    proof cases
      assume "b2∈vbits"
      show ?thesis
      proof cases
        assume "v1<v2"
        with assms show ?thesis using l i i2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
      next
        assume "¬ v1<v2"
        with assms show ?thesis using l i i2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
      qed
    next
      assume "b2∉vbits"
      with l i i2 ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    case u: (UINT b2 v2)
    then show ?thesis
    proof cases
      assume "b2∈vbits"
      show ?thesis
      proof cases
        assume "b2<b1"
        then show ?thesis
        proof cases
          assume "v1<v2"
          with assms show ?thesis using l i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ by simp
        next
          assume "¬ v1<v2"
          with assms show ?thesis using l i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b2<b1‘ by simp
        qed
      next
        assume "¬ b2<b1"
        with l i u ‘b1∈vbits‘ show ?thesis using assms by simp
      qed
    next
      assume "b2∉vbits"
      with l i u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    case (ADDRESS x3)
    with l i ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case (BALANCE x4)
    with l i ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case THIS
    with l i ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case SENDER
    with l i ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case VALUE
    with l i ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case TRUE
    with l i ‘b1∈vbits‘ show ?thesis using assms by simp
  next
    case FALSE
```

```
      with l i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (LVAL x7)
      with l i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (PLUS x81 x82)
      with l i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (MINUS x91 x92)
      with l i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (EQUAL x101 x102)
      with l i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (LESS x111 x112)
      with l i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (AND x121 x122)
      with l i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (OR x131 x132)
      with l i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (NOT x131)
      with l i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with l i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with l i ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case CONTRACTS
      with l i ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "¬ b1∈vbits"
    with l i show ?thesis using assms by simp
  qed
next
  case u: (UINT b1 v1)
  show ?thesis
  proof cases
    assume "b1∈vbits"
    show ?thesis
    proof (cases "eupdate e2")
      case i: (INT b2 v2)
      then show ?thesis
      proof cases
        let ?v="v1+v2"
        assume "b2∈vbits"
        show ?thesis
        proof cases
          assume "b1<b2"
          then show ?thesis
          proof cases
            assume "v1<v2"
            with assms show ?thesis using l i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b1<b2‘ by simp
          next
            assume "¬ v1<v2"
            with assms show ?thesis using l i u ‘b1∈vbits‘ ‘b2∈vbits‘ ‘b1<b2‘ by simp
          qed
        next
          assume "¬ b1<b2"
```

```
      with l i u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "b2∉vbits"
    with l i u ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
  case u2: (UINT b2 v2)
  then show ?thesis
  proof cases
    assume "b2∈vbits"
    show ?thesis
    proof cases
      assume "v1<v2"
      with assms show ?thesis using l u u2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
    next
      assume "¬ v1<v2"
      with assms show ?thesis using l u u2 ‘b1∈vbits‘ ‘b2∈vbits‘ by simp
    qed
  next
    assume "¬b2∈vbits"
    with l u u2 ‘b1∈vbits‘ show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (BALANCE x4)
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case THIS
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case SENDER
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case VALUE
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case TRUE
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case FALSE
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LVAL x7)
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (PLUS x81 x82)
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (MINUS x91 x92)
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (EQUAL x101 x102)
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (LESS x111 x112)
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (AND x121 x122)
  with l u ‘b1∈vbits‘ show ?thesis using assms by simp
next
  case (OR x131 x132)
```

```
      with l u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (NOT x131)
      with l u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with l u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with l u ‘b1∈vbits‘ show ?thesis using assms by simp
    next
      case CONTRACTS
      with l u ‘b1∈vbits‘ show ?thesis using assms by simp
    qed
  next
    assume "¬ b1∈vbits"
    with l u show ?thesis using assms by simp
  qed
next
  case (ADDRESS x3)
  with l show ?thesis using assms by simp
next
  case (BALANCE x4)
  with l show ?thesis using assms by simp
next
  case THIS
  with l show ?thesis using assms by simp
next
  case SENDER
  with l show ?thesis using assms by simp
next
  case VALUE
  with l show ?thesis using assms by simp
next
  case TRUE
  with l show ?thesis using assms by simp
next
  case FALSE
  with l show ?thesis using assms by simp
next
  case (LVAL x7)
  with l show ?thesis using assms by simp
next
  case (PLUS x81 x82)
  with l show ?thesis using assms by simp
next
  case (MINUS x91 x92)
  with l show ?thesis using assms by simp
next
  case (EQUAL x101 x102)
  with l show ?thesis using assms by simp
next
  case (LESS x111 x112)
  with l show ?thesis using assms by simp
next
  case (AND x121 x122)
  with l show ?thesis using assms by simp
next
  case (OR x131 x132)
  with l show ?thesis using assms by simp
next
  case (NOT x131)
  with l show ?thesis using assms by simp
next
```

```
    case (CALL x181 x182)
    with l show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with l show ?thesis using assms by simp
  next
    case CONTRACTS
    with l show ?thesis using assms by simp
  qed
next
  case a: (AND e1 e2)
  show ?thesis
  proof (cases "eupdate e1")
    case (INT x11 x12)
    with a show ?thesis using assms by simp
  next
    case (UINT x21 x22)
    with a show ?thesis using assms by simp
  next
    case (ADDRESS x3)
    with a show ?thesis using assms by simp
  next
    case (BALANCE x4)
    with a show ?thesis using assms by simp
  next
    case THIS
    with a show ?thesis using assms by simp
  next
    case SENDER
    with a show ?thesis using assms by simp
  next
    case VALUE
    with a show ?thesis using assms by simp
  next
    case t: TRUE
    show ?thesis
    proof (cases "eupdate e2")
      case (INT x11 x12)
      with a t show ?thesis using assms by simp
    next
      case (UINT x21 x22)
      with a t show ?thesis using assms by simp
    next
      case (ADDRESS x3)
      with a t show ?thesis using assms by simp
    next
      case (BALANCE x4)
      with a t show ?thesis using assms by simp
    next
      case THIS
      with a t show ?thesis using assms by simp
    next
      case SENDER
      with a t show ?thesis using assms by simp
    next
      case VALUE
      with a t show ?thesis using assms by simp
    next
      case TRUE
      with a t show ?thesis using assms by simp
    next
      case FALSE
      with a t show ?thesis using assms by simp
    next
```

```
    case (LVAL x7)
    with a t show ?thesis using assms by simp
  next
    case (PLUS x81 x82)
    with a t show ?thesis using assms by simp
  next
    case (MINUS x91 x92)
    with a t show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with a t show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with a t show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with a t show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with a t show ?thesis using assms by simp
  next
    case (NOT x131)
    with a t show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with a t show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with a t show ?thesis using assms by simp
  next
    case CONTRACTS
    with a t show ?thesis using assms by simp
  qed
next
  case f: FALSE
  show ?thesis
  proof (cases "eupdate e2")
    case (INT x11 x12)
    with a f show ?thesis using assms by simp
  next
    case (UINT x21 x22)
    with a f show ?thesis using assms by simp
  next
    case (ADDRESS x3)
    with a f show ?thesis using assms by simp
  next
    case (BALANCE x4)
    with a f show ?thesis using assms by simp
  next
    case THIS
    with a f show ?thesis using assms by simp
  next
    case SENDER
    with a f show ?thesis using assms by simp
  next
    case VALUE
    with a f show ?thesis using assms by simp
  next
    case TRUE
    with a f show ?thesis using assms by simp
  next
    case FALSE
    with a f show ?thesis using assms by simp
  next
```

```
    case (LVAL x7)
    with a f show ?thesis using assms by simp
  next
    case (PLUS x81 x82)
    with a f show ?thesis using assms by simp
  next
    case (MINUS x91 x92)
    with a f show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with a f show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with a f show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with a f show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with a f show ?thesis using assms by simp
  next
    case (NOT x131)
    with a f show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with a f show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with a f show ?thesis using assms by simp
  next
    case CONTRACTS
    with a f show ?thesis using assms by simp
  qed
next
  case (LVAL x7)
  with a show ?thesis using assms by simp
next
  case (PLUS x81 x82)
  with a show ?thesis using assms by simp
next
  case (MINUS x91 x92)
  with a show ?thesis using assms by simp
next
  case (EQUAL x101 x102)
  with a show ?thesis using assms by simp
next
  case (LESS x111 x112)
  with a show ?thesis using assms by simp
next
  case (AND x121 x122)
  with a show ?thesis using assms by simp
next
  case (OR x131 x132)
  with a show ?thesis using assms by simp
next
  case (NOT x131)
  with a show ?thesis using assms by simp
next
  case (CALL x181 x182)
  with a show ?thesis using assms by simp
next
  case (ECALL x191 x192 x193)
  with a show ?thesis using assms by simp
next
```

```
    case CONTRACTS
    with a show ?thesis using assms by simp
  qed
next
  case o: (OR e1 e2)
  show ?thesis
  proof (cases "eupdate e1")
    case (INT x11 x12)
    with o show ?thesis using assms by simp
  next
    case (UINT x21 x22)
    with o show ?thesis using assms by simp
  next
    case (ADDRESS x3)
    with o show ?thesis using assms by simp
  next
    case (BALANCE x4)
    with o show ?thesis using assms by simp
  next
    case THIS
    with o show ?thesis using assms by simp
  next
    case SENDER
    with o show ?thesis using assms by simp
  next
    case VALUE
    with o show ?thesis using assms by simp
  next
    case t: TRUE
    show ?thesis
    proof (cases "eupdate e2")
      case (INT x11 x12)
      with o t show ?thesis using assms by simp
    next
      case (UINT x21 x22)
      with o t show ?thesis using assms by simp
    next
      case (ADDRESS x3)
      with o t show ?thesis using assms by simp
    next
      case (BALANCE x4)
      with o t show ?thesis using assms by simp
    next
      case THIS
      with o t show ?thesis using assms by simp
    next
      case SENDER
      with o t show ?thesis using assms by simp
    next
      case VALUE
      with o t show ?thesis using assms by simp
    next
      case TRUE
      with o t show ?thesis using assms by simp
    next
      case FALSE
      with o t show ?thesis using assms by simp
    next
      case (LVAL x7)
      with o t show ?thesis using assms by simp
    next
      case (PLUS x81 x82)
      with o t show ?thesis using assms by simp
    next
```

```
      case (MINUS x91 x92)
      with o t show ?thesis using assms by simp
    next
      case (EQUAL x101 x102)
      with o t show ?thesis using assms by simp
    next
      case (LESS x111 x112)
      with o t show ?thesis using assms by simp
    next
      case (AND x121 x122)
      with o t show ?thesis using assms by simp
    next
      case (OR x131 x132)
      with o t show ?thesis using assms by simp
    next
      case (NOT x131)
      with o t show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with o t show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with o t show ?thesis using assms by simp
    next
      case CONTRACTS
      with o t show ?thesis using assms by simp
    qed
  next
    case f: FALSE
    show ?thesis
    proof (cases "eupdate e2")
      case (INT x11 x12)
      with o f show ?thesis using assms by simp
    next
      case (UINT x21 x22)
      with o f show ?thesis using assms by simp
    next
      case (ADDRESS x3)
      with o f show ?thesis using assms by simp
    next
      case (BALANCE x4)
      with o f show ?thesis using assms by simp
    next
      case THIS
      with o f show ?thesis using assms by simp
    next
      case SENDER
      with o f show ?thesis using assms by simp
    next
      case VALUE
      with o f show ?thesis using assms by simp
    next
      case TRUE
      with o f show ?thesis using assms by simp
    next
      case FALSE
      with o f show ?thesis using assms by simp
    next
      case (LVAL x7)
      with o f show ?thesis using assms by simp
    next
      case (PLUS x81 x82)
      with o f show ?thesis using assms by simp
    next
```

```
      case (MINUS x91 x92)
      with o f show ?thesis using assms by simp
    next
      case (EQUAL x101 x102)
      with o f show ?thesis using assms by simp
    next
      case (LESS x111 x112)
      with o f show ?thesis using assms by simp
    next
      case (AND x121 x122)
      with o f show ?thesis using assms by simp
    next
      case (OR x131 x132)
      with o f show ?thesis using assms by simp
    next
      case (NOT x131)
      with o f show ?thesis using assms by simp
    next
      case (CALL x181 x182)
      with o f show ?thesis using assms by simp
    next
      case (ECALL x191 x192 x193)
      with o f show ?thesis using assms by simp
    next
      case CONTRACTS
      with o f show ?thesis using assms by simp
    qed
  next
    case (LVAL x7)
    with o show ?thesis using assms by simp
  next
    case (PLUS x81 x82)
    with o show ?thesis using assms by simp
  next
    case (MINUS x91 x92)
    with o show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with o show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with o show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with o show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with o show ?thesis using assms by simp
  next
    case (NOT x131)
    with o show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with o show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with o show ?thesis using assms by simp
  next
    case CONTRACTS
    with o show ?thesis using assms by simp
  qed
next
  case o: (NOT x)
  show ?thesis
```

```
  proof (cases "eupdate x")
    case (INT x11 x12)
    with o show ?thesis using assms by simp
  next
    case (UINT x21 x22)
    with o show ?thesis using assms by simp
  next
    case (ADDRESS x3)
    with o show ?thesis using assms by simp
  next
    case (BALANCE x4)
    with o show ?thesis using assms by simp
  next
    case THIS
    with o show ?thesis using assms by simp
  next
    case SENDER
    with o show ?thesis using assms by simp
  next
    case VALUE
    with o show ?thesis using assms by simp
  next
    case t: TRUE
    with o show ?thesis using assms by simp
  next
    case f: FALSE
    with o show ?thesis using assms by simp
  next
    case (LVAL x7)
    with o show ?thesis using assms by simp
  next
    case (PLUS x81 x82)
    with o show ?thesis using assms by simp
  next
    case (MINUS x91 x92)
    with o show ?thesis using assms by simp
  next
    case (EQUAL x101 x102)
    with o show ?thesis using assms by simp
  next
    case (LESS x111 x112)
    with o show ?thesis using assms by simp
  next
    case (AND x121 x122)
    with o show ?thesis using assms by simp
  next
    case (OR x131 x132)
    with o show ?thesis using assms by simp
  next
    case (NOT x131)
    with o show ?thesis using assms by simp
  next
    case (CALL x181 x182)
    with o show ?thesis using assms by simp
  next
    case (ECALL x191 x192 x193)
    with o show ?thesis using assms by simp
  next
    case CONTRACTS
    with o show ?thesis using assms by simp
  qed
next
  case (CALL x181 x182)
  with assms show ?thesis by simp
```

```
next
  case (ECALL x191 x192 x193)
  with assms show ?thesis by simp
next
  case CONTRACTS
  with assms show ?thesis by simp
qed

lemma no_gas:
  assumes "¬ g > costs_ex ex env cd st"
  shows "expr ex env cd st g = Exception Gas"
proof (cases ex)
  case (INT x11 x12)
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case (UINT x21 x22)
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case (ADDRESS x3)
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case (BALANCE x4)
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case THIS
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case SENDER
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case VALUE
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case TRUE
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case FALSE
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case (LVAL x10)
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case (PLUS x111 x112)
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case (MINUS x121 x122)
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case (EQUAL x131 x132)
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case (LESS x141 x142)
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case (AND x151 x152)
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case (OR x161 x162)
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case (NOT x17)
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case (CALL x181 x182)
```

```
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case (ECALL x191 x192 x193)
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
next
  case CONTRACTS
  with assms show ?thesis by (simp add: Statements.solidity.expr.simps)
qed

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 (cases "expr e1 env cd st g")
  case s1: (n a g')
  then show ?thesis
  proof (cases a)
    case f1:(Pair a b)
    then show ?thesis
    proof (cases a)
      case k1:(KValue x1)
      then show ?thesis
      proof (cases b)
        case v1: (Value x1)
        then show ?thesis
        proof (cases "expr e2 env cd st g'")
          case s2: (n a' g'')
          then show ?thesis
          proof (cases a')
            case f2:(Pair a' b')
            then show ?thesis
            proof (cases a')
              case (KValue x1')
              with s1 f1 k1 v1 assms(1) assms(2) show ?thesis by auto
            next
              case (KCDptr x2)
              with s1 f1 k1 v1 assms(1) assms(2) show ?thesis by auto
            next
              case (KMemptr x2')
              with s1 f1 k1 v1 assms(1) assms(2) show ?thesis by auto
            next
              case (KStoptr x3')
              with s1 f1 k1 v1 assms(1) assms(2) show ?thesis by auto
            qed
          qed
        next
          case (e e)
          then show ?thesis using k1 s1 v1 assms(1) assms(2) f1 by auto
        qed
      next
        case (Calldata x2)
        then show ?thesis using k1 s1 assms(1) f1 by auto
      next
        case (Memory x2)
        then show ?thesis using k1 s1 assms(1) f1 by auto
      next
        case (Storage x3)
        then show ?thesis using k1 s1 assms(1) f1 by auto
      qed
    next
      case (KCDptr x2)
      then show ?thesis using s1 assms(1) f1 by fastforce
    next
```

```
      case (KMemptr x2)
      then show ?thesis using s1 assms(1) f1 by fastforce
    next
      case (KStoptr x3)
      then show ?thesis using s1 assms(1) f1 by fastforce
    qed
  qed
next
  case (e e)
  then show ?thesis using assms(1) by simp
qed

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 (induction ix arbitrary: tp loc env cd st g)
  case Nil
  then show ?case by simp
next
  case c1: (Cons i ix)
  then show ?case
  proof (cases tp)
    case tp1: (STArray al tp)
    then show ?thesis
    proof (cases "expr i env cd st g")
      case s1: (n a g')
      then show ?thesis
      proof (cases a)
        case f1: (Pair a b)
        then show ?thesis
        proof (cases a)
          case k1: (KValue v)
          then show ?thesis
          proof (cases b)
            case v1: (Value t)
            then show ?thesis
            proof (cases "less t (TUInt 256) v (ShowL_int al)")
              case None
              with v1 k1 tp1 s1 c1.prems f1 show ?thesis by (simp add:Statements.solidity.ssel.simps)
            next
              case s2: (Some a)
              then show ?thesis
              proof (cases a)
                case p1: (Pair a b)
                then show ?thesis
                proof (cases b)
                  case (TSInt x1)
                  with s2 p1 v1 k1 tp1 s1 c1.prems f1 show ?thesis by (simp
add:Statements.solidity.ssel.simps)
                next
                  case (TUInt x2)
                  with s2 p1 v1 k1 tp1 s1 c1.prems f1 show ?thesis by (simp
add:Statements.solidity.ssel.simps)
                next
                  case b1: TBool
                  show ?thesis
                  proof cases
                    assume "a = ShowL_bool True"
                    from c1.IH[OF c1.prems] have
                      "ssel tp (hash loc v) ix env cd st g' = ssel tp (hash loc v) (map f ix) env cd st
g'"
                      by simp
                    with mp s2 b1 p1 v1 k1 tp1 s1 c1.prems f1 show ?thesis by (simp
add:Statements.solidity.ssel.simps)
```

```
                next
                  assume "¬ a = ShowL_bool True"
                  with s2 p1 v1 k1 tp1 s1 c1.prems f1 show ?thesis by (simp
add:Statements.solidity.ssel.simps)
                qed
              next
                case TAddr
                with s2 p1 v1 k1 tp1 s1 c1.prems f1 show ?thesis by (simp
add:Statements.solidity.ssel.simps)
              qed
            qed
          qed
        next
          case (Calldata x2)
          with k1 tp1 s1 c1.prems f1 show ?thesis by (simp add:Statements.solidity.ssel.simps)
        next
          case (Memory x2)
          with k1 tp1 s1 c1.prems f1 show ?thesis by (simp add:Statements.solidity.ssel.simps)
        next
          case (Storage x3)
          with k1 tp1 s1 c1.prems f1 show ?thesis by (simp add:Statements.solidity.ssel.simps)
        qed
      next
        case (KCDptr x2)
        with tp1 s1 c1.prems f1 show ?thesis by (simp add:Statements.solidity.ssel.simps)
      next
        case (KMemptr x2)
        with tp1 s1 c1.prems f1 show ?thesis by (simp add:Statements.solidity.ssel.simps)
      next
        case (KStoptr x3)
        with tp1 s1 c1.prems f1 show ?thesis by (simp add:Statements.solidity.ssel.simps)
      qed
    qed
  next
    case (e e)
    with tp1 c1.prems show ?thesis by (simp add:Statements.solidity.ssel.simps)
  qed
next
  case tp1: (STMap _ t)
  then show ?thesis
  proof (cases "expr i env cd st g")
    case s1: (n a g')
    then show ?thesis
    proof (cases a)
      case f1: (Pair a b)
      then show ?thesis
      proof (cases a)
        case k1: (KValue v)
        from c1.IH[OF c1.prems] have
          "ssel tp (hash loc v) ix env cd st g = ssel tp (hash loc v) (map f ix) env cd st g" by simp
        with k1 tp1 s1 c1 f1 show ?thesis by (simp add:Statements.solidity.ssel.simps)
      next
        case (KCDptr x2)
        with tp1 s1 c1.prems f1 show ?thesis by (simp add:Statements.solidity.ssel.simps)
      next
        case (KMemptr x2)
        with tp1 s1 c1.prems f1 show ?thesis by (simp add:Statements.solidity.ssel.simps)
      next
        case (KStoptr x3)
        with tp1 s1 c1.prems f1 show ?thesis by (simp add:Statements.solidity.ssel.simps)
      qed
    qed
  next
    case (e e)
```

```
      with tp1 c1.prems show ?thesis by (simp add:Statements.solidity.ssel.simps)
    qed
  next
    case (STValue x2)
    then show ?thesis by (simp add:Statements.solidity.ssel.simps)
  qed
qed

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 (induction ix arbitrary: c tp loc env cd st g)
  case Nil
  then show ?case by simp
next
  case c1: (Cons i ix)
  then show ?case
  proof (cases tp)
    case tp1: (MTArray al tp)
    then show ?thesis
    proof (cases ix)
      case Nil
      thus ?thesis using tp1 c1.prems by (auto simp add:Statements.solidity.msel.simps)
    next
      case c2: (Cons a list)
      then show ?thesis
      proof (cases "expr i env cd st g")
        case s1: (n a g')
        then show ?thesis
        proof (cases a)
          case f1: (Pair a b)
          then show ?thesis
          proof (cases a)
            case k1: (KValue v)
            then show ?thesis
            proof (cases b)
              case v1: (Value t)
              then show ?thesis
              proof (cases "less t (TUInt 256) v (ShowL_int al)")
                case None
                with v1 k1 tp1 s1 c1.prems f1 show ?thesis using c2 by (simp
add:Statements.solidity.msel.simps)
              next
                case s2: (Some a)
                then show ?thesis
                proof (cases a)
                  case p1: (Pair a b)
                  then show ?thesis
                  proof (cases b)
                    case (TSInt x1)
                    with s2 p1 v1 k1 tp1 s1 c1.prems f1 show ?thesis using c2 by (simp
add:Statements.solidity.msel.simps)
                  next
                    case (TUInt x2)
                    with s2 p1 v1 k1 tp1 s1 c1.prems f1 show ?thesis using c2 by (simp
add:Statements.solidity.msel.simps)
                  next
                    case b1: TBool
                    show ?thesis
                    proof cases
                      assume "a = ShowL_bool True"
                      then show ?thesis
                      proof (cases c)
                        case True
```

```
                        then show ?thesis
                        proof (cases "accessStore (hash loc v) (memory st)")
                          case None
                          with s2 b1 p1 v1 k1 tp1 s1 c1.prems f1 True show ?thesis using c2 by (simp
add:Statements.solidity.msel.simps)
                        next
                          case s3: (Some a)
                          then show ?thesis
                          proof (cases a)
                            case (MValue x1)
                            with s2 s3 b1 p1 v1 k1 tp1 s1 c1.prems f1 True show ?thesis using c2 by
(simp add:Statements.solidity.msel.simps)
                          next
                            case mp: (MPointer l)
                            from c1.IH[OF c1.prems]
                              have "msel c tp l ix env cd st g' = msel c tp l (map f ix) env cd st g'"
by simp
                            with mp s2 s3 b1 p1 v1 k1 tp1 s1 c1.prems f1 True show ?thesis using c2
by (simp add:Statements.solidity.msel.simps)
                            qed
                          qed
                        next
                          case False
                          then show ?thesis
                          proof (cases "accessStore (hash loc v) cd")
                            case None
                            with s2 b1 p1 v1 k1 tp1 s1 c1.prems f1 False show ?thesis using c2 by (simp
add:Statements.solidity.msel.simps)
                          next
                            case s3: (Some a)
                            then show ?thesis
                            proof (cases a)
                              case (MValue x1)
                              with s2 s3 b1 p1 v1 k1 tp1 s1 c1.prems f1 False show ?thesis using c2 by
(simp add:Statements.solidity.msel.simps)
                            next
                              case mp: (MPointer l)
                              from c1.IH[OF c1.prems]
                                have "msel c tp l ix env cd st g' = msel c tp l (map f ix) env cd st g'"
by simp
                              with mp s2 s3 b1 p1 v1 k1 tp1 s1 c1.prems f1 False show ?thesis using c2
by (simp add:Statements.solidity.msel.simps)
                              qed
                            qed
                          qed
                        next
                          assume "¬ a = ShowL_bool True"
                          with s2 p1 v1 k1 tp1 s1 c1.prems f1 show ?thesis using c2 by (simp
add:Statements.solidity.msel.simps)
                        qed
                      next
                        case TAddr
                        with s2 p1 v1 k1 tp1 s1 c1.prems f1 show ?thesis using c2 by (simp
add:Statements.solidity.msel.simps)
                      qed
                    qed
                  qed
                next
                  case (Calldata x2)
                  with k1 tp1 s1 c1.prems f1 show ?thesis using c2 by (simp add:Statements.solidity.msel.simps)
                next
                  case (Memory x2)
                  with k1 tp1 s1 c1.prems f1 show ?thesis using c2 by (simp add:Statements.solidity.msel.simps)
                next
```

```
            case (Storage x3)
            with k1 tp1 s1 c1.prems f1 show ?thesis using c2 by (simp add:Statements.solidity.msel.simps)
          qed
        next
          case (KCDptr x2)
          with tp1 s1 c1.prems f1 show ?thesis using c2 by (simp add:Statements.solidity.msel.simps)
        next
          case (KMemptr x2)
          with tp1 s1 c1.prems f1 show ?thesis using c2 by (simp add:Statements.solidity.msel.simps)
        next
          case (KStoptr x3)
          with tp1 s1 c1.prems f1 show ?thesis using c2 by (simp add:Statements.solidity.msel.simps)
        qed
      qed
    next
      case (e e)
      with tp1 c1.prems show ?thesis using c2 by (simp add:Statements.solidity.msel.simps)
    qed
  qed
next
  case (MTValue x2)
  then show ?thesis by (simp add:Statements.solidity.msel.simps)
qed
qed

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 (cases "fmlookup (denvalue env) i")
  case None
  then show ?thesis by (simp add:Statements.solidity.rexp.simps)
next
  case s1: (Some a)
  then show ?thesis
  proof (cases a)
    case p1: (Pair tp b)
    then show ?thesis
    proof (cases b)
      case k1: (Stackloc l)
      then show ?thesis
      proof (cases "accessStore l (stack st)")
        case None
        with s1 p1 k1 show ?thesis by (simp add:Statements.solidity.rexp.simps)
      next
        case s2: (Some a')
        then show ?thesis
        proof (cases a')
          case (KValue _)
          with s1 s2 p1 k1 show ?thesis by (simp add:Statements.solidity.rexp.simps)
        next
          case cp: (KCDptr cp)
          then show ?thesis
          proof (cases tp)
            case (Value x1)
            with cp s1 s2 p1 k1 show ?thesis by (simp add:Statements.solidity.rexp.simps)
          next
            case mt: (Calldata ct)
            from msel_eq_msel have
              "msel False ct cp ex env cd st=msel False ct cp (map f ex) env cd st" using assms by
blast
            thus ?thesis using s1 s2 p1 k1 mt cp by (simp add:Statements.solidity.rexp.simps)
          next
            case mt: (Memory mt)
            from msel_eq_msel have
```

```
          "msel True mt cp ex env cd st=msel True mt cp (map f ex) env cd st" using assms by blast
        thus ?thesis using s1 s2 p1 k1 mt cp by (simp add:Statements.solidity.rexp.simps)
      next
        case (Storage x3)
        with cp s1 s2 p1 k1 show ?thesis by (simp add:Statements.solidity.rexp.simps)
      qed
    next
      case mp: (KMemptr mp)
      then show ?thesis
      proof (cases tp)
        case (Value x1)
        with mp s1 s2 p1 k1 show ?thesis by (simp add:Statements.solidity.rexp.simps)
      next
        case mt: (Calldata ct)
        from msel_eq_msel have
          "msel True ct mp ex env cd st=msel True ct mp (map f ex) env cd st" using assms by blast
        thus ?thesis using s1 s2 p1 k1 mt mp by (simp add:Statements.solidity.rexp.simps)
      next
        case mt: (Memory mt)
        from msel_eq_msel have
          "msel True mt mp ex env cd st=msel True mt mp (map f ex) env cd st" using assms by blast
        thus ?thesis using s1 s2 p1 k1 mt mp by (simp add:Statements.solidity.rexp.simps)
      next
        case (Storage x3)
        with mp s1 s2 p1 k1 show ?thesis by (simp add:Statements.solidity.rexp.simps)
      qed
    next
      case sp: (KStoptr sp)
      then show ?thesis
      proof (cases tp)
        case (Value x1)
        then show ?thesis using s1 s2 p1 k1 sp by (simp add:Statements.solidity.rexp.simps)
      next
        case (Calldata x2)
        then show ?thesis using s1 s2 p1 k1 sp by (simp add:Statements.solidity.rexp.simps)
      next
        case (Memory x2)
        then show ?thesis using s1 s2 p1 k1 sp by (simp add:Statements.solidity.rexp.simps)
      next
        case st: (Storage stp)
        from ssel_eq_ssel have
          "ssel stp sp ex env cd st=ssel stp sp (map f ex) env cd st" using assms by blast
        thus ?thesis using s1 s2 p1 k1 st sp by (simp add:Statements.solidity.rexp.simps)
      qed
    qed
  next
    case sl:(Storeloc sl)
    then show ?thesis
    proof (cases tp)
      case (Value x1)
      then show ?thesis using s1 p1 sl by (simp add:Statements.solidity.rexp.simps)
    next
      case (Calldata x2)
      then show ?thesis using s1 p1 sl by (simp add:Statements.solidity.rexp.simps)
    next
      case (Memory x2)
      then show ?thesis using s1 p1 sl by (simp add:Statements.solidity.rexp.simps)
    next
      case st: (Storage stp)
      from ssel_eq_ssel have
        "ssel stp sl ex env cd st=ssel stp sl (map f ex) env cd st" using assms by blast
      thus ?thesis using s1 sl p1 st by (simp add:Statements.solidity.rexp.simps)
    qed
```

```
    qed
  qed
qed
```

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 (induction ex and lv)
  case (Id x g)
  then show ?case by simp
next
  case (Ref d ix g)
  then show ?case using ref_eq[where f="eupdate"] by simp
next
  case (INT b v g)
  then show ?case
  proof (cases "g > 0")
    case True
    then show ?thesis
    proof cases
      assume "b∈vbits"
      show ?thesis
      proof cases
        let ?m_def = "(-(2^(b-1)) + (v+2^(b-1)) mod (2^b))"
        assume "v ≥ 0"

        from ‘b∈vbits‘ True have
          "expr (E.INT b v) env cd st g = Normal ((KValue (createSInt b v), Value (TSInt b)), g)"
          by (simp add:Statements.solidity.expr.simps)
        also have "createSInt b v = createSInt b ?m_def" using ‘b∈vbits‘ ‘v ≥ 0‘ unfolding
createSInt_def by auto
        also from ‘v ≥ 0‘ ‘b∈vbits‘ True have
          "Normal ((KValue (createSInt b ?m_def), Value (TSInt b)),g) = expr (eupdate (E.INT b v)) env
cd st g"
          by (simp add:Statements.solidity.expr.simps)
        finally show "expr (E.INT b v) env cd st g = expr (eupdate (E.INT b v)) env cd st g" by simp
      next
        let ?m_def = "(2^(b-1) - (-v+2^(b-1)-1) mod (2^b) - 1)"
        assume "¬ v ≥ 0"

        from ‘b∈vbits‘ True have
          "expr (E.INT b v) env cd st g = Normal ((KValue (createSInt b v), Value (TSInt b)), g)" by
(simp add:Statements.solidity.expr.simps)
        also have "createSInt b v = createSInt b ?m_def" using ‘b∈vbits‘ ‘¬ v ≥ 0‘ unfolding
createSInt_def by auto
        also from ‘¬ v ≥ 0‘ ‘b∈vbits‘ True have
          "Normal ((KValue (createSInt b ?m_def), Value (TSInt b)),g) =expr (eupdate (E.INT b v)) env cd
st g"
          by (simp add:Statements.solidity.expr.simps)
        finally show "expr (E.INT b v) env cd st g = expr (eupdate (E.INT b v)) env cd st g" by simp
      qed
    next
      assume "¬ b∈vbits"
      thus ?thesis by auto
    qed
  next
    case False
    then show ?thesis using no_gas by simp
  qed
next
  case (UINT x1 x2)
  then show ?case by (simp add:Statements.solidity.expr.simps createUInt_def)
next
```

```
  case (ADDRESS x)
  then show ?case by simp
next
  case (BALANCE x)
  then show ?case by simp
next
  case THIS
  then show ?case by simp
next
  case SENDER
  then show ?case by simp
next
  case VALUE
  then show ?case by simp
next
  case TRUE
  then show ?case by simp
next
  case FALSE
  then show ?case by simp
next
  case (LVAL x)
  then show ?case by (simp add:Statements.solidity.expr.simps createUInt_def)
next
  case p: (PLUS e1 e2 g)
  show ?case
  proof (cases "eupdate e1")
    case i: (INT b1 v1)
    with p.IH have expr1: "expr e1 env cd st g = expr (E.INT b1 v1) env cd st g" by simp
    then show ?thesis
    proof (cases "g > 0")
      case True
      then show ?thesis
      proof (cases)
        assume "b1 ∈ vbits"
        with expr1 True
          have "expr e1 env cd st g = Normal ((KValue (createSInt b1 v1), Value (TSInt b1)), g)" by
(simp add:Statements.solidity.expr.simps createSInt_def)
        moreover from i ‘b1 ∈ vbits‘
          have "v1 < 2^(b1-1)" and "v1 ≥ -(2^(b1-1))" using update_bounds_int by auto
        moreover from ‘b1 ∈ vbits‘ have "0 < b1" by auto
        ultimately have r1: "expr e1 env cd st g = Normal ((KValue (ShowL_int v1), Value (TSInt b1)),g)"
          using createSInt_id[of v1 b1] by simp
        thus ?thesis
        proof (cases "eupdate e2")
          case i2: (INT b2 v2)
          with p.IH have expr2: "expr e2 env cd st g = expr (E.INT b2 v2) env cd st g" by simp
          then show ?thesis
          proof (cases)
            let ?v="v1 + v2"
            assume "b2 ∈ vbits"
            with expr2 True
              have "expr e2 env cd st g = Normal ((KValue (createSInt b2 v2), Value (TSInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
            moreover from i2 ‘b2 ∈ vbits‘
              have "v2 < 2^(b2-1)" and "v2 ≥ -(2^(b2-1))" using update_bounds_int by auto
            moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
            ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TSInt
b2)),g)"
              using createSInt_id[of v2 b2] by simp
            thus ?thesis
            proof (cases)
              let ?x="- (2 ^ (max b1 b2 - 1)) + (?v + 2 ^ (max b1 b2 - 1)) mod 2 ^ max b1 b2"
              assume "?v≥0"
```

```
            hence "createSInt (max b1 b2) ?v = (ShowL_int ?x)" by (simp add: createSInt_def)
            moreover have "add (TSInt b1) (TSInt b2) (ShowL_int v1) (ShowL_int v2)
              = Some (createSInt (max b1 b2) ?v, TSInt (max b1 b2))"
              using Read_ShowL_id add_def olift.simps(1)[of "(+)" b1 b2] by simp
            ultimately have "expr (PLUS e1 e2) env cd st g
              = Normal ((KValue (ShowL_int ?x), Value (TSInt (max b1 b2))),g)" using r1 r2 True by
(simp add:Statements.solidity.expr.simps)
            moreover have "expr (eupdate (PLUS e1 e2)) env cd st g
              = Normal ((KValue (ShowL_int ?x), Value (TSInt (max b1 b2))),g)"
            proof -
              from ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ ‘?v≥0‘
                have "eupdate (PLUS e1 e2) = E.INT (max b1 b2) ?x" using i i2 by simp
              moreover have "expr (E.INT (max b1 b2) ?x) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt (max b1 b2))),g)"
              proof -
                from ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ have "max b1 b2 ∈ vbits" using vbits_max by simp
                with True have "expr (E.INT (max b1 b2) ?x) env cd st g
                  = Normal ((KValue (createSInt (max b1 b2) ?x), Value (TSInt (max b1 b2))),g)" by
(simp add:Statements.solidity.expr.simps)
                moreover from ‘0 < b1‘
                  have "?x < 2 ^ (max b1 b2 - 1)" using upper_bound3 by simp
                moreover from ‘0 < b1‘ have "0 < max b1 b2" using max_def by simp
                ultimately show ?thesis using createSInt_id[of ?x "max b1 b2"] by simp
              qed
              ultimately show ?thesis by simp
            qed
            ultimately show ?thesis by simp
          next
            let ?x="2^(max b1 b2 -1) - (-?v+2^(max b1 b2-1)-1) mod (2^max b1 b2) - 1"
            assume "¬ ?v≥0"
            hence "createSInt (max b1 b2) ?v = (ShowL_int ?x)" unfolding createSInt_def by simp
            moreover have "add (TSInt b1) (TSInt b2) (ShowL_int v1) (ShowL_int v2)
              = Some (createSInt (max b1 b2) ?v, TSInt (max b1 b2))"
              using Read_ShowL_id add_def olift.simps(1)[of "(+)" b1 b2] by simp
            ultimately have "expr (PLUS e1 e2) env cd st g
              = Normal ((KValue (ShowL_int ?x), Value (TSInt (max b1 b2))),g)" using True r1 r2 by
(simp add:Statements.solidity.expr.simps)
            moreover have "expr (eupdate (PLUS e1 e2)) env cd st g
              = Normal ((KValue (ShowL_int ?x), Value (TSInt (max b1 b2))),g)"
            proof -
              from ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ ‘¬?v≥0‘
                have "eupdate (PLUS e1 e2) = E.INT (max b1 b2) ?x" using i i2 by simp
              moreover have "expr (E.INT (max b1 b2) ?x) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt (max b1 b2))),g)"
              proof -
                from ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ have "max b1 b2 ∈ vbits" using vbits_max by simp
                with True have "expr (E.INT (max b1 b2) ?x) env cd st g
                  = Normal ((KValue (createSInt (max b1 b2) ?x), Value (TSInt (max b1 b2))),g)" by
(simp add:Statements.solidity.expr.simps)
                moreover from ‘0 < b1‘
                  have "?x ≥ - (2 ^ (max b1 b2 - 1))" using lower_bound2[of "max b1 b2" ?v] by simp
                moreover from ‘b1 > 0‘ have "2^(max b1 b2 -1) > (0::nat)" by simp
                  hence "2^(max b1 b2 -1) - (-?v+2^(max b1 b2-1)-1) mod (2^max b1 b2) - 1 < 2 ^ (max
b1 b2 - 1)"
                    by (simp add: algebra_simps flip: zle_diff1_eq)
                moreover from ‘0 < b1‘ have "0 < max b1 b2" using max_def by simp
                ultimately show ?thesis using createSInt_id[of ?x "max b1 b2"] by simp
              qed
              ultimately show ?thesis by simp
            qed
            ultimately show ?thesis by simp
          qed
        next
          assume "¬ b2 ∈ vbits"
```

```
          with p i i2 show ?thesis by (simp add:Statements.solidity.expr.simps)
        qed
      next
        case u2: (UINT b2 v2)
        with p.IH have expr2: "expr e2 env cd st g = expr (UINT b2 v2) env cd st g" by simp
        then show ?thesis
        proof (cases)
          let ?v="v1 + v2"
          assume "b2 ∈ vbits"
          with expr2 True
            have "expr e2 env cd st g = Normal ((KValue (createUInt b2 v2), Value (TUInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
          moreover from u2 ‘b2 ∈ vbits‘
            have "v2 < 2^b2" and "v2 ≥ 0" using update_bounds_uint by auto
          moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
          ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TUInt
b2)),g)"
            using createUInt_id[of v2 b2] by simp
          thus ?thesis
          proof (cases)
            assume "b2<b1"
            thus ?thesis
            proof (cases)
              let ?x="- (2 ^ (b1 - 1)) + (?v + 2 ^ (b1 - 1)) mod 2 ^ b1"
              assume "?v≥0"
              hence "createSInt b1 ?v = (ShowL_int ?x)" using ‘b2<b1‘ unfolding createSInt_def by
auto
              moreover have "add (TSInt b1) (TUInt b2) (ShowL_int v1) (ShowL_int v2)
                = Some (createSInt b1 ?v, TSInt b1)"
                using Read_ShowL_id add_def olift.simps(3)[of "(+)" b1 b2] ‘b2<b1‘ by simp
              ultimately have "expr (PLUS e1 e2) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b1)),g)" using r1 r2 True by (simp
add:Statements.solidity.expr.simps)
              moreover have "expr (eupdate (PLUS e1 e2)) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b1)),g)"
              proof -
                from ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ ‘?v≥0‘ ‘b2<b1‘
                  have "eupdate (PLUS e1 e2) = E.INT b1 ?x" using i u2 by simp
                moreover have "expr (E.INT b1 ?x) env cd st g
                  = Normal ((KValue (ShowL_int ?x), Value (TSInt b1)),g)"
                proof -
                  from ‘b1 ∈ vbits‘ True have "expr (E.INT b1 ?x) env cd st g
                    = Normal ((KValue (createSInt b1 ?x), Value (TSInt b1)),g)" by (simp
add:Statements.solidity.expr.simps)
                  moreover from ‘0 < b1‘ have "?x < 2 ^ (b1 - 1)" using upper_bound2 by simp
                  ultimately show ?thesis using createSInt_id[of ?x "b1"] ‘0 < b1‘ by simp
                qed
                ultimately show ?thesis by simp
              qed
              ultimately show ?thesis by simp
            next
              let ?x="2^(b1 -1) - (-?v+2^(b1-1)-1) mod (2^b1) - 1"
              assume "¬ ?v≥0"
              hence "createSInt b1 ?v = (ShowL_int ?x)" unfolding createSInt_def by simp
              moreover have "add (TSInt b1) (TUInt b2) (ShowL_int v1) (ShowL_int v2)
                = Some (createSInt b1 ?v, TSInt b1)"
                using Read_ShowL_id add_def olift.simps(3)[of "(+)" b1 b2] ‘b2<b1‘ by simp
              ultimately have "expr (PLUS e1 e2) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b1)),g)" using r1 r2 True by (simp
add:Statements.solidity.expr.simps)
              moreover have "expr (eupdate (PLUS e1 e2)) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b1)),g)"
              proof -
                from ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ ‘¬?v≥0‘ ‘b2<b1‘
```

```
                have "eupdate (PLUS e1 e2) = E.INT b1 ?x" using i u2 by simp
              moreover have "expr (E.INT b1 ?x) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b1)),g)"
              proof -
                from ‘b1 ∈ vbits‘ True have "expr (E.INT b1 ?x) env cd st g
                  = Normal ((KValue (createSInt b1 ?x), Value (TSInt b1)),g)" by (simp
add:Statements.solidity.expr.simps)
                moreover from ‘0 < b1‘ have "?x ≥ - (2 ^ (b1 - 1))" using upper_bound2 by simp
                moreover have "2^(b1-1) - (-?v+2^(b1-1)-1) mod (2^b1) - 1 < 2 ^ (b1 - 1)"
                  by (simp add: algebra_simps flip: int_one_le_iff_zero_less)
                ultimately show ?thesis using createSInt_id[of ?x b1] ‘0 < b1‘ by simp
              qed
              ultimately show ?thesis by simp
            qed
            ultimately show ?thesis by simp
          qed
        next
          assume "¬ b2 < b1"
          with i u2 have "eupdate (PLUS e1 e2) = PLUS (eupdate e1) (eupdate e2)" by simp
          with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
        qed
      next
        assume "¬ b2 ∈ vbits"
        with p i u2 show ?thesis by (simp add:Statements.solidity.expr.simps)
      qed
    next
      case (ADDRESS _)
      with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (BALANCE _)
      with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case THIS
      with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case SENDER
      with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case VALUE
      with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case TRUE
      with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case FALSE
      with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (LVAL _)
      with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (PLUS _ _)
      with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (MINUS _ _)
      with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (EQUAL _ _)
      with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (LESS _ _)
      with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (AND _ _)
      with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
```

```
        next
          case (OR _ _)
          with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (NOT _)
          with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (CALL x181 x182)
          with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (ECALL x191 x192 x193)
          with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case CONTRACTS
          with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
        qed
      next
        assume "¬ b1 ∈ vbits"
        with p i show ?thesis by (simp add:Statements.solidity.expr.simps)
      qed
    next
      case False
      then show ?thesis using no_gas by simp
    qed
  next
    case u: (UINT b1 v1)
    with p.IH have expr1: "expr e1 env cd st g = expr (UINT b1 v1) env cd st g" by simp
    then show ?thesis
    proof (cases "g > 0")
      case True
      then show ?thesis
      proof (cases)
        assume "b1 ∈ vbits"
        with expr1 True
          have "expr e1 env cd st g = Normal ((KValue (createUInt b1 v1), Value (TUInt b1)),g)" by
(simp add:Statements.solidity.expr.simps)
        moreover from u ‘b1 ∈ vbits‘
          have "v1 < 2^b1" and "v1 ≥ 0" using update_bounds_uint by auto
        moreover from ‘b1 ∈ vbits‘ have "0 < b1" by auto
        ultimately have r1: "expr e1 env cd st g = Normal ((KValue (ShowL_int v1), Value (TUInt b1)),g)"
          by (simp add:Statements.solidity.expr.simps createUInt_def)
        thus ?thesis
        proof (cases "eupdate e2")
          case u2: (UINT b2 v2)
          with p.IH have expr2: "expr e2 env cd st g = expr (UINT b2 v2) env cd st g" by simp
          then show ?thesis
          proof (cases)
            let ?v="v1 + v2"
            let ?x="?v mod 2 ^ max b1 b2"
            assume "b2 ∈ vbits"
            with expr2 True
              have "expr e2 env cd st g = Normal ((KValue (createUInt b2 v2), Value (TUInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
            moreover from u2 ‘b2 ∈ vbits‘
              have "v2 < 2^b2" and "v2 ≥ 0" using update_bounds_uint by auto
            moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
            ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TUInt
b2)),g)"
              by (simp add:Statements.solidity.expr.simps createUInt_def)
            moreover have "add (TUInt b1) (TUInt b2) (ShowL_int v1) (ShowL_int v2)
              = Some (createUInt (max b1 b2) ?v, TUInt (max b1 b2))"
              using Read_ShowL_id add_def olift.simps(2)[of "(+)" b1 b2] by simp
            ultimately have "expr (PLUS e1 e2) env cd st g
              = Normal ((KValue (ShowL_int ?x), Value (TUInt (max b1 b2))),g)" using r1 True by (simp
```

```
add:Statements.solidity.expr.simps createUInt_def)
            moreover have "expr (eupdate (PLUS e1 e2)) env cd st g
              = Normal ((KValue (ShowL_int ?x), Value (TUInt (max b1 b2))),g)"
            proof -
              from ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘
                have "eupdate (PLUS e1 e2) = UINT (max b1 b2) ?x" using u u2 by simp
              moreover have "expr (UINT (max b1 b2) ?x) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TUInt (max b1 b2))),g)"
              proof -
                from ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ have "max b1 b2 ∈ vbits" using vbits_max by simp
                with True have "expr (UINT (max b1 b2) ?x) env cd st g
                  = Normal ((KValue (createUInt (max b1 b2) ?x), Value (TUInt (max b1 b2))),g)" by
(simp add:Statements.solidity.expr.simps)
                moreover from ‘0 < b1‘
                  have "?x < 2 ^ (max b1 b2)" by simp
                moreover from ‘0 < b1‘ have "0 < max b1 b2" using max_def by simp
                ultimately show ?thesis by (simp add:Statements.solidity.expr.simps createUInt_def)
              qed
              ultimately show ?thesis by simp
            qed
            ultimately show ?thesis by simp
          next
            assume "¬ b2 ∈ vbits"
            with p u u2 show ?thesis by (simp add:Statements.solidity.expr.simps)
          qed
        next
          case i2: (INT b2 v2)
          with p.IH have expr2: "expr e2 env cd st g = expr (E.INT b2 v2) env cd st g" by simp
          then show ?thesis
          proof (cases)
            let ?v="v1 + v2"
            assume "b2 ∈ vbits"
            with expr2 True
              have "expr e2 env cd st g = Normal ((KValue (createSInt b2 v2), Value (TSInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
            moreover from i2 ‘b2 ∈ vbits‘
              have "v2 < 2^(b2-1)" and "v2 ≥ -(2^(b2-1))" using update_bounds_int by auto
            moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
            ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TSInt
b2)),g)"
              using createSInt_id[of v2 b2] by simp
            thus ?thesis
            proof (cases)
              assume "b1<b2"
              thus ?thesis
              proof (cases)
                let ?x="- (2 ^ (b2 - 1)) + (?v + 2 ^ (b2 - 1)) mod 2 ^ b2"
                assume "?v≥0"
                hence "createSInt b2 ?v = (ShowL_int ?x)" using ‘b1<b2‘ unfolding createSInt_def by
auto
                moreover have "add (TUInt b1) (TSInt b2) (ShowL_int v1) (ShowL_int v2)
                  = Some (createSInt b2 ?v, TSInt b2)"
                  using Read_ShowL_id add_def olift.simps(4)[of "(+)" b1 b2] ‘b1<b2‘ by simp
                ultimately have "expr (PLUS e1 e2) env cd st g
                  = Normal ((KValue (ShowL_int ?x), Value (TSInt b2)),g)" using r1 r2 True by (simp
add:Statements.solidity.expr.simps)
                moreover have "expr (eupdate (PLUS e1 e2)) env cd st g
                  = Normal ((KValue (ShowL_int ?x), Value (TSInt b2)),g)"
                proof -
                  from ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ ‘?v≥0‘ ‘b1<b2‘
                    have "eupdate (PLUS e1 e2) = E.INT b2 ?x" using u i2 by simp
                  moreover have "expr (E.INT b2 ?x) env cd st g
                    = Normal ((KValue (ShowL_int ?x), Value (TSInt b2)),g)"
                  proof -
```

```
                  from ‘b2 ∈ vbits‘ True have "expr (E.INT b2 ?x) env cd st g
                    = Normal ((KValue (createSInt b2 ?x), Value (TSInt b2)),g)" by (simp
add:Statements.solidity.expr.simps)
                  moreover from ‘0 < b2‘ have "?x < 2 ^ (b2 - 1)" using upper_bound2 by simp
                  ultimately show ?thesis using createSInt_id[of ?x "b2"] ‘0 < b2‘ by simp
                qed
                ultimately show ?thesis by simp
              qed
              ultimately show ?thesis by simp
            next
              let ?x="2^(b2 -1) - (-?v+2^(b2-1)-1) mod (2^b2) - 1"
              assume "¬ ?v≥0"
              hence "createSInt b2 ?v = (ShowL_int ?x)" unfolding createSInt_def by simp
              moreover have "add (TUInt b1) (TSInt b2) (ShowL_int v1) (ShowL_int v2)
                = Some (createSInt b2 ?v, TSInt b2)"
                using Read_ShowL_id add_def olift.simps(4)[of "(+)" b1 b2] ‘b1<b2‘ by simp
              ultimately have "expr (PLUS e1 e2) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b2)),g)" using r1 r2 True by (simp
add:Statements.solidity.expr.simps)
              moreover have "expr (eupdate (PLUS e1 e2)) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b2)),g)"
              proof -
                from ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ ‘¬?v≥0‘ ‘b1<b2‘
                  have "eupdate (PLUS e1 e2) = E.INT b2 ?x" using u i2 by simp
                moreover have "expr (E.INT b2 ?x) env cd st g
                  = Normal ((KValue (ShowL_int ?x), Value (TSInt b2)),g)"
                proof -
                  from ‘b2 ∈ vbits‘ True have "expr (E.INT b2 ?x) env cd st g
                    = Normal ((KValue (createSInt b2 ?x), Value (TSInt b2)),g)" by (simp
add:Statements.solidity.expr.simps)
                  moreover from ‘0 < b2‘ have "?x ≥ - (2 ^ (b2 - 1))" using upper_bound2 by simp
                  moreover have "2^(b2-1) - (-?v+2^(b2-1)-1) mod (2^b2) - 1 < 2 ^ (b2 - 1)"
                    by (simp add: algebra_simps flip: int_one_le_iff_zero_less)
                  ultimately show ?thesis using createSInt_id[of ?x b2] ‘0 < b2‘ by simp
                qed
                ultimately show ?thesis by simp
              qed
              ultimately show ?thesis by simp
            qed
          next
            assume "¬ b1 < b2"
            with p u i2 show ?thesis by (simp add:Statements.solidity.expr.simps)
          qed
        next
          assume "¬ b2 ∈ vbits"
          with p u i2 show ?thesis by (simp add:Statements.solidity.expr.simps)
        qed
      next
        case (ADDRESS _)
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (BALANCE _)
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case THIS
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case SENDER
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case VALUE
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case TRUE
```

```
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case FALSE
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (LVAL _)
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (PLUS _ _)
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (MINUS _ _)
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (EQUAL _ _)
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (LESS _ _)
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (AND _ _)
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (OR _ _)
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (NOT _)
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (CALL x181 x182)
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (ECALL x191 x192 x193)
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case CONTRACTS
        with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
      qed
    next
      assume "¬ b1 ∈ vbits"
      with p u show ?thesis by (simp add:Statements.solidity.expr.simps)
    qed
  next
    case False
    then show ?thesis using no_gas by simp
  qed
next
  case (ADDRESS x3)
  with p show ?thesis by (simp add:Statements.solidity.expr.simps)
next
  case (BALANCE x4)
  with p show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
next
  case THIS
  with p show ?thesis by (simp add:Statements.solidity.expr.simps)
next
  case SENDER
  with p show ?thesis by (simp add:Statements.solidity.expr.simps)
next
  case VALUE
  with p show ?thesis by (simp add:Statements.solidity.expr.simps)
next
  case TRUE
```

```
    with p show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case FALSE
    with p show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (LVAL x7)
    with p show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (PLUS x81 x82)
    with p show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (MINUS x91 x92)
    with p show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (EQUAL x101 x102)
    with p show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (LESS x111 x112)
    with p show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (AND x121 x122)
    with p show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (OR x131 x132)
    with p show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (NOT x131)
    with p show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (CALL x181 x182)
    with p show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (ECALL x191 x192 x193)
    with p show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case CONTRACTS
    with p show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  qed
next
  case m: (MINUS e1 e2 g)
  show ?case
  proof (cases "eupdate e1")
    case i: (INT b1 v1)
    with m.IH have expr1: "expr e1 env cd st g = expr (E.INT b1 v1) env cd st g" by simp
    then show ?thesis
    proof (cases "g > 0")
      case True
      show ?thesis
      proof (cases)
        assume "b1 ∈ vbits"
        with expr1 True
          have "expr e1 env cd st g = Normal ((KValue (createSInt b1 v1), Value (TSInt b1)),g)" by
(simp add:Statements.solidity.expr.simps)
```

```
        moreover from i ‘b1 ∈ vbits‘
          have "v1 < 2^(b1-1)" and "v1 ≥ -(2^(b1-1))" using update_bounds_int by auto
        moreover from ‘b1 ∈ vbits‘ have "0 < b1" by auto
        ultimately have r1: "expr e1 env cd st g = Normal ((KValue (ShowL_int v1), Value (TSInt b1)),g)"
          using createSInt_id[of v1 b1] by simp
        thus ?thesis
        proof (cases "eupdate e2")
          case i2: (INT b2 v2)
          with m.IH have expr2: "expr e2 env cd st g = expr (E.INT b2 v2) env cd st g" by simp
          then show ?thesis
          proof (cases)
            let ?v="v1 - v2"
            assume "b2 ∈ vbits"
            with expr2 True
              have "expr e2 env cd st g = Normal ((KValue (createSInt b2 v2), Value (TSInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
            moreover from i2 ‘b2 ∈ vbits‘
              have "v2 < 2^(b2-1)" and "v2 ≥ -(2^(b2-1))" using update_bounds_int by auto
            moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
            ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TSInt
b2)),g)"
              using createSInt_id[of v2 b2] by simp

            from ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ have
              u_def: "eupdate (MINUS e1 e2) =
              (let v = v1 - v2
                in if 0 ≤ v
                  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))"
              using i i2 eupdate.simps(11)[of e1 e2] by simp

            show ?thesis
            proof (cases)
              let ?x="- (2 ^ (max b1 b2 - 1)) + (?v + 2 ^ (max b1 b2 - 1)) mod 2 ^ max b1 b2"
              assume "?v≥0"
              hence "createSInt (max b1 b2) ?v = (ShowL_int ?x)" unfolding createSInt_def by simp
              moreover have "sub (TSInt b1) (TSInt b2) (ShowL_int v1) (ShowL_int v2)
                = Some (createSInt (max b1 b2) ?v, TSInt (max b1 b2))"
                using Read_ShowL_id sub_def olift.simps(1)[of "(-)" b1 b2] by simp
              ultimately have "expr (MINUS e1 e2) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt (max b1 b2))),g)" using r1 r2 True by
(simp add:Statements.solidity.expr.simps)
              moreover have "expr (eupdate (MINUS e1 e2)) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt (max b1 b2))),g)"
              proof -
                from u_def have "eupdate (MINUS e1 e2) = E.INT (max b1 b2) ?x" using ‘?v≥0‘ by simp
                moreover have "expr (E.INT (max b1 b2) ?x) env cd st g
                  = Normal ((KValue (ShowL_int ?x), Value (TSInt (max b1 b2))),g)"
                proof -
                  from ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ have "max b1 b2 ∈ vbits" using vbits_max by simp
                  with True have "expr (E.INT (max b1 b2) ?x) env cd st g
                    = Normal ((KValue (createSInt (max b1 b2) ?x), Value (TSInt (max b1 b2))),g)" by
(simp add:Statements.solidity.expr.simps)
                  moreover from ‘0 < b1‘
                    have "?x < 2 ^ (max b1 b2 - 1)" using upper_bound2 by simp
                  moreover from ‘0 < b1‘ have "0 < max b1 b2" using max_def by simp
                  ultimately show ?thesis using createSInt_id[of ?x "max b1 b2"] by simp
                qed
                ultimately show ?thesis by simp
              qed
              ultimately show ?thesis by simp
            next
```

```
            let ?x="2^(max b1 b2 -1) - (-?v+2^(max b1 b2-1)-1) mod (2^max b1 b2) - 1"
            assume "¬ ?v≥0"
            hence "createSInt (max b1 b2) ?v = (ShowL_int ?x)" unfolding createSInt_def by simp
            moreover have "sub (TSInt b1) (TSInt b2) (ShowL_int v1) (ShowL_int v2)
              = Some (createSInt (max b1 b2) ?v, TSInt (max b1 b2))"
              using Read_ShowL_id sub_def olift.simps(1)[of "(-)" b1 b2] by simp
            ultimately have "expr (MINUS e1 e2) env cd st g
              = Normal ((KValue (ShowL_int ?x), Value (TSInt (max b1 b2))),g)" using r1 r2 True by
(simp add:Statements.solidity.expr.simps)
            moreover have "expr (eupdate (MINUS e1 e2)) env cd st g
              = Normal ((KValue (ShowL_int ?x), Value (TSInt (max b1 b2))),g)"
            proof -
              from u_def have "eupdate (MINUS e1 e2) = E.INT (max b1 b2) ?x" using ‘¬ ?v≥0‘ by
simp
              moreover have "expr (E.INT (max b1 b2) ?x) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt (max b1 b2))),g)"
              proof -
                from ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ have "max b1 b2 ∈ vbits" using vbits_max by simp
                with True have "expr (E.INT (max b1 b2) ?x) env cd st g
                  = Normal ((KValue (createSInt (max b1 b2) ?x), Value (TSInt (max b1 b2))),g)" by
(simp add:Statements.solidity.expr.simps createSInt_def)
                moreover from ‘0 < b1‘
                  have "?x ≥ - (2 ^ (max b1 b2 - 1))" using lower_bound2[of "max b1 b2" ?v] by simp
                moreover from ‘b1 > 0‘ have "2^(max b1 b2 -1) > (0::nat)" by simp
                  hence "2^(max b1 b2 -1) - (-?v+2^(max b1 b2-1)-1) mod (2^max b1 b2) - 1 < 2 ^ (max
b1 b2 - 1)"
                  by (simp add: algebra_simps flip: int_one_le_iff_zero_less)
                moreover from ‘0 < b1‘ have "0 < max b1 b2" using max_def by simp
                ultimately show ?thesis using createSInt_id[of ?x "max b1 b2"] by simp
              qed
              ultimately show ?thesis by simp
            qed
            ultimately show ?thesis by simp
          qed
        next
          assume "¬ b2 ∈ vbits"
          with m i i2 show ?thesis by (simp add:Statements.solidity.expr.simps)
        qed
      next
        case u: (UINT b2 v2)
        with m.IH have expr2: "expr e2 env cd st g = expr (UINT b2 v2) env cd st g" by simp
        then show ?thesis
        proof (cases)
          let ?v="v1 - v2"
          assume "b2 ∈ vbits"
          with expr2 True
            have "expr e2 env cd st g = Normal ((KValue (createUInt b2 v2), Value (TUInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
          moreover from u ‘b2 ∈ vbits‘
            have "v2 < 2^b2" and "v2 ≥ 0" using update_bounds_uint by auto
          moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
          ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TUInt
b2)),g)"
            using createUInt_id[of v2 b2] by simp
          thus ?thesis
          proof (cases)
            assume "b2<b1"
            with ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ have
              u_def: "eupdate (MINUS e1 e2) =
              (let v = v1 - v2
                in if 0 ≤ v
                  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))"
              using i u eupdate.simps(11)[of e1 e2] by simp
```

```
            show ?thesis
            proof (cases)
              let ?x="- (2 ^ (b1 - 1)) + (?v + 2 ^ (b1 - 1)) mod 2 ^ b1"
              assume "?v≥0"
              hence "createSInt b1 ?v = (ShowL_int ?x)" using ‘b2<b1‘ unfolding createSInt_def by
auto
              moreover have "sub (TSInt b1) (TUInt b2) (ShowL_int v1) (ShowL_int v2)
                = Some (createSInt b1 ?v, TSInt b1)"
                using Read_ShowL_id sub_def olift.simps(3)[of "(-)" b1 b2] ‘b2<b1‘ by simp
              ultimately have "expr (MINUS e1 e2) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b1)),g)" using r1 r2 True by (simp
add:Statements.solidity.expr.simps)
              moreover have "expr (eupdate (MINUS e1 e2)) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b1)),g)"
              proof -
                from u_def have "eupdate (MINUS e1 e2) = E.INT b1 ?x" using ‘?v≥0‘ by simp
                moreover have "expr (E.INT b1 ?x) env cd st g
                  = Normal ((KValue (ShowL_int ?x), Value (TSInt b1)),g)"
                proof -
                  from ‘b1 ∈ vbits‘ True have "expr (E.INT b1 ?x) env cd st g
                    = Normal ((KValue (createSInt b1 ?x), Value (TSInt b1)),g)" by (simp
add:Statements.solidity.expr.simps)
                  moreover from ‘0 < b1‘ have "?x < 2 ^ (b1 - 1)" using upper_bound2 by simp
                  ultimately show ?thesis using createSInt_id[of ?x "b1"] ‘0 < b1‘ by simp
                qed
                ultimately show ?thesis by simp
              qed
              ultimately show ?thesis by simp
            next
              let ?x="2^(b1 -1) - (-?v+2^(b1-1)-1) mod (2^b1) - 1"
              assume "¬ ?v≥0"
              hence "createSInt b1 ?v = (ShowL_int ?x)" unfolding createSInt_def by simp
              moreover have "sub (TSInt b1) (TUInt b2) (ShowL_int v1) (ShowL_int v2)
                = Some (createSInt b1 ?v, TSInt b1)"
                using Read_ShowL_id sub_def olift.simps(3)[of "(-)" b1 b2] ‘b2<b1‘ by simp
              ultimately have "expr (MINUS e1 e2) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b1)),g)" using r1 r2 True by (simp
add:Statements.solidity.expr.simps)
              moreover have "expr (eupdate (MINUS e1 e2)) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b1)),g)"
              proof -
                from u_def have "eupdate (MINUS e1 e2) = E.INT b1 ?x" using ‘¬ ?v≥0‘ by simp
                moreover have "expr (E.INT b1 ?x) env cd st g
                  = Normal ((KValue (ShowL_int ?x), Value (TSInt b1)),g)"
                proof -
                  from ‘b1 ∈ vbits‘ True have "expr (E.INT b1 ?x) env cd st g
                    = Normal ((KValue (createSInt b1 ?x), Value (TSInt b1)),g)" by (simp
add:Statements.solidity.expr.simps)
                  moreover from ‘0 < b1‘ have "?x ≥ - (2 ^ (b1 - 1))" using upper_bound2 by simp
                  moreover have "2^(b1-1) - (-?v+2^(b1-1)-1) mod (2^b1) - 1 < 2 ^ (b1 - 1)"
                    by (simp add: algebra_simps flip: int_one_le_iff_zero_less)
                  ultimately show ?thesis using createSInt_id[of ?x b1] ‘0 < b1‘ by simp
                qed
                ultimately show ?thesis by simp
              qed
              ultimately show ?thesis by simp
            qed
          next
            assume "¬ b2 < b1"
            with i u have "eupdate (MINUS e1 e2) = MINUS (eupdate e1) (eupdate e2)" by simp
            with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
          qed
        next
          assume "¬ b2 ∈ vbits"
```

```
        with m i u show ?thesis by (simp add:Statements.solidity.expr.simps)
      qed
    next
      case (ADDRESS _)
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (BALANCE _)
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case THIS
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case SENDER
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case VALUE
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case TRUE
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case FALSE
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (LVAL _)
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (PLUS _ _)
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (MINUS _ _)
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (EQUAL _ _)
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (LESS _ _)
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (AND _ _)
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (OR _ _)
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (NOT _)
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (CALL x181 x182)
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (ECALL x191 x192 x193)
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case CONTRACTS
      with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
    qed
  next
    assume "¬ b1 ∈ vbits"
    with m i show ?thesis by (simp add:Statements.solidity.expr.simps)
  qed
next
  case False
  then show ?thesis using no_gas by simp
```

```
    qed
  next
    case u: (UINT b1 v1)
    with m.IH have expr1: "expr e1 env cd st g = expr (UINT b1 v1) env cd st g" by simp
    then show ?thesis
    proof (cases "g > 0")
      case True
      show ?thesis
      proof (cases)
        assume "b1 ∈ vbits"
        with expr1 True
          have "expr e1 env cd st g = Normal ((KValue (createUInt b1 v1), Value (TUInt b1)),g)" by
(simp add:Statements.solidity.expr.simps)
        moreover from u `b1 ∈ vbits`
          have "v1 < 2^b1" and "v1 ≥ 0" using update_bounds_uint by auto
        moreover from `b1 ∈ vbits` have "0 < b1" by auto
        ultimately have r1: "expr e1 env cd st g = Normal ((KValue (ShowL_int v1), Value (TUInt b1)),g)"
          by (simp add:Statements.solidity.expr.simps createUInt_def)
        thus ?thesis
        proof (cases "eupdate e2")
          case u2: (UINT b2 v2)
          with m.IH have expr2: "expr e2 env cd st g = expr (UINT b2 v2) env cd st g" by simp
          then show ?thesis
          proof (cases)
            let ?v="v1 - v2"
            let ?x="?v mod 2 ^ max b1 b2"
            assume "b2 ∈ vbits"
            with expr2 True
              have "expr e2 env cd st g = Normal ((KValue (createUInt b2 v2), Value (TUInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
            moreover from u2 `b2 ∈ vbits`
              have "v2 < 2^b2" and "v2 ≥ 0" using update_bounds_uint by auto
            moreover from `b2 ∈ vbits` have "0 < b2" by auto
            ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TUInt
b2)),g)"
              by (simp add:Statements.solidity.expr.simps createUInt_def)
            moreover have "sub (TUInt b1) (TUInt b2) (ShowL_int v1) (ShowL_int v2)
              = Some (createUInt (max b1 b2) ?v, TUInt (max b1 b2))"
              using Read_ShowL_id sub_def olift.simps(2)[of "(-)" b1 b2] by simp
            ultimately have "expr (MINUS e1 e2) env cd st g
              = Normal ((KValue (ShowL_int ?x), Value (TUInt (max b1 b2))),g)" using r1 True by (simp
add:Statements.solidity.expr.simps createUInt_def)
            moreover have "expr (eupdate (MINUS e1 e2)) env cd st g
              = Normal ((KValue (ShowL_int ?x), Value (TUInt (max b1 b2))),g)"
            proof -
              from `b1 ∈ vbits` `b2 ∈ vbits`
                have "eupdate (MINUS e1 e2) = UINT (max b1 b2) ?x" using u u2 by simp
              moreover have "expr (UINT (max b1 b2) ?x) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TUInt (max b1 b2))),g)"
              proof -
                from `b1 ∈ vbits` `b2 ∈ vbits` have "max b1 b2 ∈ vbits" using vbits_max by simp
                with True have "expr (UINT (max b1 b2) ?x) env cd st g
                  = Normal ((KValue (createUInt (max b1 b2) ?x), Value (TUInt (max b1 b2))),g)" by
(simp add:Statements.solidity.expr.simps)
                moreover from `0 < b1`
                  have "?x < 2 ^ (max b1 b2)" by simp
                moreover from `0 < b1` have "0 < max b1 b2" using max_def by simp
                ultimately show ?thesis by (simp add:Statements.solidity.expr.simps createUInt_def)
              qed
              ultimately show ?thesis by simp
            qed
            ultimately show ?thesis by simp
          next
            assume "¬ b2 ∈ vbits"
```

```
          with m u u2 show ?thesis by (simp add:Statements.solidity.expr.simps)
        qed
      next
        case i: (INT b2 v2)
        with m.IH have expr2: "expr e2 env cd st g = expr (E.INT b2 v2) env cd st g" by simp
        then show ?thesis
        proof (cases)
          let ?v="v1 - v2"
          assume "b2 ∈ vbits"
          with expr2 True
            have "expr e2 env cd st g = Normal ((KValue (createSInt b2 v2), Value (TSInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
          moreover from i ‘b2 ∈ vbits‘
            have "v2 < 2^(b2-1)" and "v2 ≥ -(2^(b2-1))" using update_bounds_int by auto
          moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
          ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TSInt
b2)),g)"
            using createSInt_id[of v2 b2] by simp
          thus ?thesis
          proof (cases)
            assume "b1<b2"
            with ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ have
              u_def: "eupdate (MINUS e1 e2) =
              (let v = v1 - v2
                in if 0 ≤ v
                  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))"
              using u i by simp
            show ?thesis
            proof (cases)
              let ?x="- (2 ^ (b2 - 1)) + (?v + 2 ^ (b2 - 1)) mod 2 ^ b2"
              assume "?v≥0"
              hence "createSInt b2 ?v = (ShowL_int ?x)" using ‘b1<b2‘ unfolding createSInt_def by
auto
              moreover have "sub (TUInt b1) (TSInt b2) (ShowL_int v1) (ShowL_int v2)
                = Some (createSInt b2 ?v, TSInt b2)"
                using Read_ShowL_id sub_def olift.simps(4)[of "(-)" b1 b2] ‘b1<b2‘ by simp
              ultimately have "expr (MINUS e1 e2) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b2)),g)" using r1 r2 True by (simp
add:Statements.solidity.expr.simps)
              moreover have "expr (eupdate (MINUS e1 e2)) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b2)),g)"
              proof -
                from u_def have "eupdate (MINUS e1 e2) = E.INT b2 ?x" using ‘?v≥0‘ by simp
                moreover have "expr (E.INT b2 ?x) env cd st g
                  = Normal ((KValue (ShowL_int ?x), Value (TSInt b2)),g)"
                proof -
                  from ‘b2 ∈ vbits‘ True have "expr (E.INT b2 ?x) env cd st g
                    = Normal ((KValue (createSInt b2 ?x), Value (TSInt b2)),g)" by (simp
add:Statements.solidity.expr.simps)
                  moreover from ‘0 < b2‘ have "?x < 2 ^ (b2 - 1)" using upper_bound2 by simp
                  ultimately show ?thesis using createSInt_id[of ?x "b2"] ‘0 < b2‘ by simp
                qed
                ultimately show ?thesis by simp
              qed
              ultimately show ?thesis by simp
            next
              let ?x="2^(b2 -1) - (-?v+2^(b2-1)-1) mod (2^b2) - 1"
              assume "¬ ?v≥0"
              hence "createSInt b2 ?v = (ShowL_int ?x)" unfolding createSInt_def by simp
              moreover have "sub (TUInt b1) (TSInt b2) (ShowL_int v1) (ShowL_int v2)
                = Some (createSInt b2 ?v, TSInt b2)"
                using Read_ShowL_id sub_def olift.simps(4)[of "(-)" b1 b2] ‘b1<b2‘ by simp
              ultimately have "expr (MINUS e1 e2) env cd st g
```

```
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b2)),g)" using r1 r2 True by (simp
add:Statements.solidity.expr.simps)
              moreover have "expr (eupdate (MINUS e1 e2)) env cd st g
                = Normal ((KValue (ShowL_int ?x), Value (TSInt b2)),g)"
              proof -
                from u_def have "eupdate (MINUS e1 e2) = E.INT b2 ?x" using ‘¬ ?v≥0‘ by simp
                moreover have "expr (E.INT b2 ?x) env cd st g
                  = Normal ((KValue (ShowL_int ?x), Value (TSInt b2)),g)"
                proof -
                  from ‘b2 ∈ vbits‘ True have "expr (E.INT b2 ?x) env cd st g
                    = Normal ((KValue (createSInt b2 ?x), Value (TSInt b2)),g)" by (simp
add:Statements.solidity.expr.simps)
                  moreover from ‘0 < b2‘ have "?x ≥ - (2 ^ (b2 - 1))" using upper_bound2 by simp
                  moreover have "2^(b2-1) - (-?v+2^(b2-1)-1) mod (2^b2) - 1 < 2 ^ (b2 - 1)"
                    by (simp add: algebra_simps flip: int_one_le_iff_zero_less)
                  ultimately show ?thesis using createSInt_id[of ?x b2] ‘0 < b2‘ by simp
                qed
                ultimately show ?thesis by simp
              qed
              ultimately show ?thesis by simp
            qed
          next
            assume "¬ b1 < b2"
            with m u i show ?thesis by (simp add:Statements.solidity.expr.simps)
          qed
        next
          assume "¬ b2 ∈ vbits"
          with m u i show ?thesis by (simp add:Statements.solidity.expr.simps)
        qed
      next
        case (ADDRESS _)
        with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (BALANCE _)
        with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case THIS
        with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case SENDER
        with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case VALUE
        with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case TRUE
        with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case FALSE
        with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (LVAL _)
        with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (PLUS _ _)
        with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (MINUS _ _)
        with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (EQUAL _ _)
        with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (LESS _ _)
```

```
          with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (AND _ _)
          with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (OR _ _)
          with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (NOT _)
          with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (CALL x181 x182)
          with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (ECALL x191 x192 x193)
          with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case CONTRACTS
          with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
        qed
      next
        assume "¬ b1 ∈ vbits"
        with m u show ?thesis by (simp add:Statements.solidity.expr.simps)
      qed
    next
      case False
      then show ?thesis using no_gas by simp
    qed
  next
    case (ADDRESS x3)
    with m show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (BALANCE x4)
    with m show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case THIS
    with m show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case SENDER
    with m show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case VALUE
    with m show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case TRUE
    with m show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case FALSE
    with m show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (LVAL x7)
    with m show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (PLUS x81 x82)
    with m show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (MINUS x91 x92)
    with m show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (EQUAL x101 x102)
```

```
    with m show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (LESS x111 x112)
    with m show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (AND x121 x122)
    with m show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (OR x131 x132)
    with m show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (NOT x131)
    with m show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (CALL x181 x182)
    with m show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (simp
add:Statements.solidity.expr.simps)
  next
    case (ECALL x191 x192 x193)
    with m show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (simp
add:Statements.solidity.expr.simps)
  next
    case CONTRACTS
    with m show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  qed
next
  case e: (EQUAL e1 e2 g)
  show ?case
  proof (cases "eupdate e1")
    case i: (INT b1 v1)
    with e.IH have expr1: "expr e1 env cd st g = expr (E.INT b1 v1) env cd st g" by simp
    then show ?thesis
    proof (cases "g > 0")
      case True
      then show ?thesis
      proof (cases)
        assume "b1 ∈ vbits"
        with expr1 True
          have "expr e1 env cd st g = Normal ((KValue (createSInt b1 v1), Value (TSInt b1)),g)" by
(simp add:Statements.solidity.expr.simps)
        moreover from i ‘b1 ∈ vbits‘
          have "v1 < 2^(b1-1)" and "v1 ≥ -(2^(b1-1))" using update_bounds_int by auto
        moreover from ‘b1 ∈ vbits‘ have "0 < b1" by auto
        ultimately have r1: "expr e1 env cd st g = Normal ((KValue (ShowL_int v1), Value (TSInt b1)),g)"
          using createSInt_id[of v1 b1] by simp
        thus ?thesis
        proof (cases "eupdate e2")
          case i2: (INT b2 v2)
          with e.IH have expr2: "expr e2 env cd st g = expr (E.INT b2 v2) env cd st g" by simp
          then show ?thesis
          proof (cases)
            assume "b2 ∈ vbits"
            with expr2 True
              have "expr e2 env cd st g = Normal ((KValue (createSInt b2 v2), Value (TSInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
            moreover from i2 ‘b2 ∈ vbits‘
              have "v2 < 2^(b2-1)" and "v2 ≥ -(2^(b2-1))" using update_bounds_int by auto
            moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
```

```
            ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TSInt
b2)),g)"
              using createSInt_id[of v2 b2] by simp
            with r1 True have "expr (EQUAL e1 e2) env cd st g =
              Normal ((KValue (createBool ((ReadL_int (ShowL_int v1))=((ReadL_int (ShowL_int v2))))), Value
TBool),g)"
              using equal_def plift.simps(1)[of "(=)"] by (simp add:Statements.solidity.expr.simps)
            hence "expr (EQUAL e1 e2) env cd st g = Normal ((KValue (createBool (v1=v2)), Value
TBool),g)"
              using Read_ShowL_id by simp
            with ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ True show ?thesis using i i2 by (simp
add:Statements.solidity.expr.simps createBool_def)
          next
            assume "¬ b2 ∈ vbits"
            with e i i2 show ?thesis by (simp add:Statements.solidity.expr.simps)
          qed
        next
          case u: (UINT b2 v2)
          with e.IH have expr2: "expr e2 env cd st g = expr (UINT b2 v2) env cd st g" by simp
          then show ?thesis
          proof (cases)
            assume "b2 ∈ vbits"
            with expr2 True
              have "expr e2 env cd st g = Normal ((KValue (createUInt b2 v2), Value (TUInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
            moreover from u ‘b2 ∈ vbits‘
              have "v2 < 2^b2" and "v2 ≥ 0" using update_bounds_uint by auto
            moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
            ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TUInt
b2)),g)"
              using createUInt_id[of v2 b2] by simp
            thus ?thesis
            proof (cases)
              assume "b2<b1"
              with r1 r2 True have "expr (EQUAL e1 e2) env cd st g =
                Normal ((KValue (createBool ((ReadL_int (ShowL_int v1))=((ReadL_int (ShowL_int v2))))), Value
TBool),g)"
                using equal_def plift.simps(3)[of "(=)"] by (simp add:Statements.solidity.expr.simps)
              hence "expr (EQUAL e1 e2) env cd st g = Normal ((KValue (createBool (v1=v2)), Value
TBool),g)"
                using Read_ShowL_id by simp
              with ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ ‘b2<b1‘ True show ?thesis using i u by (simp
add:Statements.solidity.expr.simps createBool_def)
            next
              assume "¬ b2 < b1"
              with i u have "eupdate (EQUAL e1 e2) = EQUAL (eupdate e1) (eupdate e2)" by simp
              with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
            qed
          next
            assume "¬ b2 ∈ vbits"
            with e i u show ?thesis by (simp add:Statements.solidity.expr.simps)
          qed
        next
          case (ADDRESS _)
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (BALANCE _)
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case THIS
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case SENDER
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
```

```
        next
          case VALUE
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case TRUE
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case FALSE
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (LVAL _)
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (PLUS _ _)
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (MINUS _ _)
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (EQUAL _ _)
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (LESS _ _)
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (AND _ _)
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (OR _ _)
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (NOT _)
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (CALL x181 x182)
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (ECALL x191 x192 x193)
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case CONTRACTS
          with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
        qed
      next
        assume "¬ b1 ∈ vbits"
        with e i show ?thesis by (simp add:Statements.solidity.expr.simps)
      qed
    next
      case False
      then show ?thesis using no_gas by simp
    qed
  next
    case u: (UINT b1 v1)
    with e.IH have expr1: "expr e1 env cd st g = expr (UINT b1 v1) env cd st g" by simp
    then show ?thesis
    proof (cases "g > 0")
      case True
      then show ?thesis
      proof (cases)
        assume "b1 ∈ vbits"
        with expr1 True
          have "expr e1 env cd st g = Normal ((KValue (createUInt b1 v1), Value (TUInt b1)),g)" by
(simp add:Statements.solidity.expr.simps)
        moreover from u ‘b1 ∈ vbits‘
```

```
          have "v1 < 2^b1" and "v1 ≥ 0" using update_bounds_uint by auto
        moreover from ‘b1 ∈ vbits‘ have "0 < b1" by auto
        ultimately have r1: "expr e1 env cd st g = Normal ((KValue (ShowL_int v1), Value (TUInt b1)),g)"
          by (simp add:Statements.solidity.expr.simps createUInt_def)
        thus ?thesis
        proof (cases "eupdate e2")
          case u2: (UINT b2 v2)
          with e.IH have expr2: "expr e2 env cd st g = expr (UINT b2 v2) env cd st g" by simp
          then show ?thesis
          proof (cases)
            assume "b2 ∈ vbits"
            with expr2 True
              have "expr e2 env cd st g = Normal ((KValue (createUInt b2 v2), Value (TUInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
            moreover from u2 ‘b2 ∈ vbits‘
              have "v2 < 2^b2" and "v2 ≥ 0" using update_bounds_uint by auto
            moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
            ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TUInt
b2)),g)"
              by (simp add:Statements.solidity.expr.simps createUInt_def)
            with r1 True have "expr (EQUAL e1 e2) env cd st g =
              Normal ((KValue (createBool ((ReadL_int (ShowL_int v1))=((ReadL_int (ShowL_int v2))))), Value
TBool),g)"
              using equal_def plift.simps(2)[of "(=)"] by (simp add:Statements.solidity.expr.simps
createUInt_def)
            hence "expr (EQUAL e1 e2) env cd st g = Normal ((KValue (createBool (v1=v2)), Value
TBool),g)"
              using Read_ShowL_id by simp
            with ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ show ?thesis using u u2 True by (simp
add:Statements.solidity.expr.simps createBool_def)
          next
            assume "¬ b2 ∈ vbits"
            with e u u2 show ?thesis by (simp add:Statements.solidity.expr.simps)
          qed
        next
          case i: (INT b2 v2)
          with e.IH have expr2: "expr e2 env cd st g = expr (E.INT b2 v2) env cd st g" by simp
          then show ?thesis
          proof (cases)
            let ?v="v1 + v2"
            assume "b2 ∈ vbits"
            with expr2 True
              have "expr e2 env cd st g = Normal ((KValue (createSInt b2 v2), Value (TSInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
            moreover from i ‘b2 ∈ vbits‘
              have "v2 < 2^(b2-1)" and "v2 ≥ -(2^(b2-1))" using update_bounds_int by auto
            moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
            ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TSInt
b2)),g)"
              using createSInt_id[of v2 b2] by simp
            thus ?thesis
            proof (cases)
              assume "b1<b2"
              with r1 r2 True have "expr (EQUAL e1 e2) env cd st g =
                Normal ((KValue (createBool ((ReadL_int (ShowL_int v1))=((ReadL_int (ShowL_int v2))))), Value
TBool),g)"
                using equal_def plift.simps(4)[of "(=)"] by (simp add:Statements.solidity.expr.simps)
              hence "expr (EQUAL e1 e2) env cd st g = Normal ((KValue (createBool (v1=v2)), Value
TBool),g)"
                using Read_ShowL_id by simp
              with ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ ‘b1<b2‘ True show ?thesis using u i by (simp
add:Statements.solidity.expr.simps createBool_def)
            next
              assume "¬ b1 < b2"
```

```
        with e u i show ?thesis by (simp add:Statements.solidity.expr.simps)
      qed
    next
      assume "¬ b2 ∈ vbits"
      with e u i show ?thesis by (simp add:Statements.solidity.expr.simps)
    qed
  next
    case (ADDRESS _)
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (BALANCE _)
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case THIS
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case SENDER
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case VALUE
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case TRUE
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case FALSE
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (LVAL _)
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (PLUS _ _)
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (MINUS _ _)
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (EQUAL _ _)
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (LESS _ _)
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (AND _ _)
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (OR _ _)
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (NOT _)
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (CALL x181 x182)
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (ECALL x191 x192 x193)
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case CONTRACTS
    with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
  qed
next
  assume "¬ b1 ∈ vbits"
  with e u show ?thesis by (simp add:Statements.solidity.expr.simps)
```

```
      qed
    next
      case False
      then show ?thesis using no_gas by simp
    qed
  next
    case (ADDRESS x3)
    with e show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (BALANCE x4)
    with e show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case THIS
    with e show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case SENDER
    with e show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case VALUE
    with e show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case TRUE
    with e show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case FALSE
    with e show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (LVAL x7)
    with e show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (PLUS x81 x82)
    with e show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (MINUS x91 x92)
    with e show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (EQUAL x101 x102)
    with e show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (LESS x111 x112)
    with e show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (AND x121 x122)
    with e show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (OR x131 x132)
    with e show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (NOT x131)
    with e show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (CALL x181 x182)
    with e show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (simp
add:Statements.solidity.expr.simps)
  next
```

```
    case (ECALL x191 x192 x193)
    with e show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (simp
add:Statements.solidity.expr.simps)
  next
    case CONTRACTS
    with e show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  qed
next
  case l: (LESS e1 e2)
  show ?case
  proof (cases "eupdate e1")
    case i: (INT b1 v1)
    with l.IH have expr1: "expr e1 env cd st g = expr (E.INT b1 v1) env cd st g" by (simp
add:Statements.solidity.expr.simps)
    then show ?thesis
    proof (cases "g > 0")
      case True
      then show ?thesis
      proof (cases)
        assume "b1 ∈ vbits"
        with expr1 True
          have "expr e1 env cd st g =Normal ((KValue (createSInt b1 v1), Value (TSInt b1)),g)" by (simp
add:Statements.solidity.expr.simps)
        moreover from i ‘b1 ∈ vbits‘
          have "v1 < 2^(b1-1)" and "v1 ≥ -(2^(b1-1))" using update_bounds_int by auto
        moreover from ‘b1 ∈ vbits‘ have "0 < b1" by auto
        ultimately have r1: "expr e1 env cd st g = Normal ((KValue (ShowL_int v1), Value (TSInt b1)),g)"
          using createSInt_id[of v1 b1] by (simp add:Statements.solidity.expr.simps)
        thus ?thesis
        proof (cases "eupdate e2")
          case i2: (INT b2 v2)
          with l.IH have expr2: "expr e2 env cd st g = expr (E.INT b2 v2) env cd st g" by simp
          then show ?thesis
          proof (cases)
            assume "b2 ∈ vbits"
            with expr2 True
              have "expr e2 env cd st g = Normal ((KValue (createSInt b2 v2), Value (TSInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
            moreover from i2 ‘b2 ∈ vbits‘
              have "v2 < 2^(b2-1)" and "v2 ≥ -(2^(b2-1))" using update_bounds_int by auto
            moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
            ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TSInt
b2)),g)"
              using createSInt_id[of v2 b2] by simp
            with r1 True have "expr (LESS e1 e2) env cd st g =
              Normal ((KValue (createBool ((ReadL_int (ShowL_int v1))<((ReadL_int (ShowL_int v2))))), Value
TBool),g)"
              using less_def plift.simps(1)[of "(<)"] by (simp add:Statements.solidity.expr.simps)
            hence "expr (LESS e1 e2) env cd st g = Normal ((KValue (createBool (v1<v2)), Value
TBool),g)"
              using Read_ShowL_id by simp
            with ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ show ?thesis using i i2 True by (simp
add:Statements.solidity.expr.simps createBool_def)
          next
            assume "¬ b2 ∈ vbits"
            with l i i2 show ?thesis by (simp add:Statements.solidity.expr.simps)
          qed
        next
          case u: (UINT b2 v2)
          with l.IH have expr2: "expr e2 env cd st g = expr (UINT b2 v2) env cd st g" by simp
          then show ?thesis
          proof (cases)
            assume "b2 ∈ vbits"
```

```
            with expr2 True
              have "expr e2 env cd st g = Normal ((KValue (createUInt b2 v2), Value (TUInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
            moreover from u ‘b2 ∈ vbits‘
              have "v2 < 2^b2" and "v2 ≥ 0" using update_bounds_uint by auto
            moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
            ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TUInt
b2)),g)"
              using createUInt_id[of v2 b2] by simp
            thus ?thesis
            proof (cases)
              assume "b2<b1"
              with r1 r2 True have "expr (LESS e1 e2) env cd st g =
                Normal ((KValue (createBool ((ReadL_int (ShowL_int v1))<((ReadL_int (ShowL_int v2))))), Value
TBool),g)"
                using less_def plift.simps(3)[of "(<)"] by (simp add:Statements.solidity.expr.simps)
              hence "expr (LESS e1 e2) env cd st g = Normal ((KValue (createBool (v1<v2)), Value
TBool),g)"
                using Read_ShowL_id by simp
              with ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ ‘b2<b1‘ show ?thesis using i u True by (simp
add:Statements.solidity.expr.simps createBool_def)
            next
              assume "¬ b2 < b1"
              with i u have "eupdate (LESS e1 e2) = LESS (eupdate e1) (eupdate e2)" by simp
              with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
            qed
          next
            assume "¬ b2 ∈ vbits"
            with l i u show ?thesis by (simp add:Statements.solidity.expr.simps)
          qed
        next
          case (ADDRESS _)
          with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (BALANCE _)
          with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case THIS
          with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case SENDER
          with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case VALUE
          with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case TRUE
          with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case FALSE
          with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (LVAL _)
          with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (PLUS _ _)
          with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (MINUS _ _)
          with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
          case (EQUAL _ _)
          with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
        next
```

```
        case (LESS _ _)
        with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (AND _ _)
        with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (OR _ _)
        with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (NOT _)
        with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (CALL x181 x182)
        with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (ECALL x191 x192 x193)
        with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case CONTRACTS
        with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
      qed
    next
      assume "¬ b1 ∈ vbits"
      with l i show ?thesis by (simp add:Statements.solidity.expr.simps)
    qed
  next
    case False
    then show ?thesis using no_gas by (simp add:Statements.solidity.expr.simps)
  qed
next
  case u: (UINT b1 v1)
  with l.IH have expr1: "expr e1 env cd st g = expr (UINT b1 v1) env cd st g" by simp
  then show ?thesis
  proof (cases "g > 0")
    case True
    then show ?thesis
    proof (cases)
      assume "b1 ∈ vbits"
      with expr1 True
        have "expr e1 env cd st g = Normal ((KValue (createUInt b1 v1), Value (TUInt b1)),g)" by
(simp add:Statements.solidity.expr.simps)
      moreover from u ‘b1 ∈ vbits‘
        have "v1 < 2^b1" and "v1 ≥ 0" using update_bounds_uint by auto
      moreover from ‘b1 ∈ vbits‘ have "0 < b1" by auto
      ultimately have r1: "expr e1 env cd st g = Normal ((KValue (ShowL_int v1), Value (TUInt b1)),g)"
        by (simp add:Statements.solidity.expr.simps createUInt_def)
      thus ?thesis
      proof (cases "eupdate e2")
        case u2: (UINT b2 v2)
        with l.IH have expr2: "expr e2 env cd st g = expr (UINT b2 v2) env cd st g" by simp
        then show ?thesis
        proof (cases)
          assume "b2 ∈ vbits"
          with expr2 True
            have "expr e2 env cd st g = Normal ((KValue (createUInt b2 v2), Value (TUInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
          moreover from u2 ‘b2 ∈ vbits‘
            have "v2 < 2^b2" and "v2 ≥ 0" using update_bounds_uint by auto
          moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
          ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TUInt
b2)),g)"
            by (simp add:Statements.solidity.expr.simps createUInt_def)
          with r1 True have "expr (LESS e1 e2) env cd st g = Normal ((KValue (createBool ((ReadL_int
(ShowL_int v1))<((ReadL_int (ShowL_int v2))))), Value TBool),g)" using less_def plift.simps(2)[of "(<)"] by
```

```
(simp add:Statements.solidity.expr.simps)
          hence "expr (LESS e1 e2) env cd st g = Normal ((KValue (createBool (v1<v2)), Value
TBool),g)" using Read_ShowL_id by simp
          with ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ show ?thesis using u u2 True by (simp
add:Statements.solidity.expr.simps createBool_def)
        next
          assume "¬ b2 ∈ vbits"
          with l u u2 show ?thesis by (simp add:Statements.solidity.expr.simps)
        qed
      next
        case i: (INT b2 v2)
        with l.IH have expr2: "expr e2 env cd st g = expr (E.INT b2 v2) env cd st g" by simp
        then show ?thesis
        proof (cases)
          let ?v="v1 + v2"
          assume "b2 ∈ vbits"
          with expr2 True
            have "expr e2 env cd st g = Normal ((KValue (createSInt b2 v2), Value (TSInt b2)),g)" by
(simp add:Statements.solidity.expr.simps)
          moreover from i ‘b2 ∈ vbits‘
            have "v2 < 2^(b2-1)" and "v2 ≥ -(2^(b2-1))" using update_bounds_int by auto
          moreover from ‘b2 ∈ vbits‘ have "0 < b2" by auto
          ultimately have r2: "expr e2 env cd st g = Normal ((KValue (ShowL_int v2), Value (TSInt
b2)),g)"
            using createSInt_id[of v2 b2] by simp
          thus ?thesis
          proof (cases)
            assume "b1<b2"
            with r1 r2 True have "expr (LESS e1 e2) env cd st g =
              Normal ((KValue (createBool ((ReadL_int (ShowL_int v1))<((ReadL_int (ShowL_int v2))))), Value
TBool),g)"
              using less_def plift.simps(4)[of "(<)"] by (simp add:Statements.solidity.expr.simps)
            hence "expr (LESS e1 e2) env cd st g = Normal ((KValue (createBool (v1<v2)), Value
TBool),g)"
              using Read_ShowL_id by simp
            with ‘b1 ∈ vbits‘ ‘b2 ∈ vbits‘ ‘b1<b2‘ show ?thesis using u i True by (simp
add:Statements.solidity.expr.simps createBool_def)
          next
            assume "¬ b1 < b2"
            with l u i show ?thesis by (simp add:Statements.solidity.expr.simps)
          qed
        next
          assume "¬ b2 ∈ vbits"
          with l u i show ?thesis by (simp add:Statements.solidity.expr.simps)
        qed
      next
        case (ADDRESS _)
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (BALANCE _)
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case THIS
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case SENDER
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case VALUE
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case TRUE
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
```

```
        case FALSE
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (LVAL _)
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (PLUS _ _)
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (MINUS _ _)
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (EQUAL _ _)
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (LESS _ _)
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (AND _ _)
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (OR _ _)
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (NOT _)
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (CALL x181 x182)
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case (ECALL x191 x192 x193)
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      next
        case CONTRACTS
        with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
      qed
    next
      assume "¬ b1 ∈ vbits"
      with l u show ?thesis by (simp add:Statements.solidity.expr.simps)
    qed
  next
    case False
    then show ?thesis using no_gas by simp
  qed
next
  case (ADDRESS x3)
  with l show ?thesis by (simp add:Statements.solidity.expr.simps)
next
  case (BALANCE x4)
  with l show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
next
  case THIS
  with l show ?thesis by (simp add:Statements.solidity.expr.simps)
next
  case SENDER
  with l show ?thesis by (simp add:Statements.solidity.expr.simps)
next
  case VALUE
  with l show ?thesis by (simp add:Statements.solidity.expr.simps)
next
  case TRUE
  with l show ?thesis by (simp add:Statements.solidity.expr.simps)
next
```

```
    case FALSE
    with l show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (LVAL x7)
    with l show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (PLUS x81 x82)
    with l show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (MINUS x91 x92)
    with l show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (EQUAL x101 x102)
    with l show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (LESS x111 x112)
    with l show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (AND x121 x122)
    with l show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (OR x131 x132)
    with l show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (NOT x131)
    with l show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (CALL x181 x182)
    with l show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (simp
add:Statements.solidity.expr.simps)
  next
    case (ECALL x191 x192 x193)
    with l show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (simp
add:Statements.solidity.expr.simps)
  next
    case CONTRACTS
    with l show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  qed
next
  case a: (AND e1 e2)
  show ?case
  proof (cases "eupdate e1")
    case (INT x11 x12)
    with a show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (UINT x21 x22)
    with a show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (ADDRESS x3)
    with a show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (BALANCE x4)
    with a show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
```

```
  case THIS
  with a show ?thesis by (simp add:Statements.solidity.expr.simps)
next
  case SENDER
  with a show ?thesis by (simp add:Statements.solidity.expr.simps)
next
  case VALUE
  with a show ?thesis by (simp add:Statements.solidity.expr.simps)
next
  case t: TRUE
  show ?thesis
  proof (cases "eupdate e2")
    case (INT x11 x12)
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (UINT x21 x22)
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (ADDRESS x3)
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (BALANCE x4)
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case THIS
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case SENDER
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case VALUE
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case TRUE
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps vtand.simps)
  next
    case FALSE
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps vtand.simps)
  next
    case (LVAL x7)
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (PLUS x81 x82)
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (MINUS x91 x92)
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (EQUAL x101 x102)
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (LESS x111 x112)
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (AND x121 x122)
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (OR x131 x132)
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (NOT x131)
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (CALL x181 x182)
```

```
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (ECALL x191 x192 x193)
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case CONTRACTS
    with a t show ?thesis by (simp add:Statements.solidity.expr.simps)
  qed
next
  case f: FALSE
  show ?thesis
  proof (cases "eupdate e2")
    case (INT b v)
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (UINT b v)
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (ADDRESS x3)
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (BALANCE x4)
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case THIS
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case SENDER
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case VALUE
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case TRUE
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps vtand.simps)
  next
    case FALSE
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps vtand.simps)
  next
    case (LVAL x7)
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (PLUS x81 x82)
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (MINUS x91 x92)
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (EQUAL x101 x102)
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (LESS x111 x112)
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (AND x121 x122)
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (OR x131 x132)
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (NOT x131)
    with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (CALL x181 x182)
```

```
      with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (ECALL x191 x192 x193)
      with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case CONTRACTS
      with a f show ?thesis by (simp add:Statements.solidity.expr.simps)
    qed
  next
    case (LVAL x7)
    with a show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case p: (PLUS x81 x82)
    with a show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (MINUS x91 x92)
    with a show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (EQUAL x101 x102)
    with a show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (LESS x111 x112)
    with a show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (AND x121 x122)
    with a show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (OR x131 x132)
    with a show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (NOT x131)
    with a show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (CALL x181 x182)
    with a show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (simp
add:Statements.solidity.expr.simps)
  next
    case (ECALL x191 x192 x193)
    with a show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (simp
add:Statements.solidity.expr.simps)
  next
    case CONTRACTS
    with a show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  qed
next
  case o: (OR e1 e2)
  show ?case
  proof (cases "eupdate e1")
    case (INT x11 x12)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (UINT x21 x22)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (ADDRESS x3)
```

```
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (BALANCE x4)
    with o show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case THIS
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case SENDER
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case VALUE
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case t: TRUE
    show ?thesis
    proof (cases "eupdate e2")
      case (INT x11 x12)
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (UINT x21 x22)
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (ADDRESS x3)
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (BALANCE x4)
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case THIS
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case SENDER
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case VALUE
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case TRUE
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps vtor.simps)
    next
      case FALSE
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps vtor.simps)
    next
      case (LVAL x7)
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (PLUS x81 x82)
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (MINUS x91 x92)
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (EQUAL x101 x102)
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (LESS x111 x112)
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (AND x121 x122)
      with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (OR x131 x132)
```

```
    with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (NOT x131)
    with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (CALL x181 x182)
    with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (ECALL x191 x192 x193)
    with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case CONTRACTS
    with o t show ?thesis by (simp add:Statements.solidity.expr.simps)
  qed
next
  case f: FALSE
  show ?thesis
  proof (cases "eupdate e2")
    case (INT b v)
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (UINT b v)
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (ADDRESS x3)
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (BALANCE x4)
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case THIS
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case SENDER
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case VALUE
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case TRUE
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps vtor.simps)
  next
    case FALSE
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps vtor.simps)
  next
    case (LVAL x7)
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (PLUS x81 x82)
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (MINUS x91 x92)
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (EQUAL x101 x102)
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (LESS x111 x112)
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (AND x121 x122)
    with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (OR x131 x132)
```

```
      with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (NOT x131)
      with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (CALL x181 x182)
      with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case (ECALL x191 x192 x193)
      with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
    next
      case CONTRACTS
      with o f show ?thesis by (simp add:Statements.solidity.expr.simps)
    qed
  next
    case (LVAL x7)
    with o show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case p: (PLUS x81 x82)
    with o show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (MINUS x91 x92)
    with o show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (EQUAL x101 x102)
    with o show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (LESS x111 x112)
    with o show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (AND x121 x122)
    with o show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (OR x131 x132)
    with o show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (NOT x131)
    with o show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  next
    case (CALL x181 x182)
    with o show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (simp
add:Statements.solidity.expr.simps)
  next
    case (ECALL x191 x192 x193)
    with o show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (simp
add:Statements.solidity.expr.simps)
  next
    case CONTRACTS
    with o show ?thesis using lift_eq[of e1 env cd st g "eupdate e1" e2 "eupdate e2"] by (auto simp
add:Statements.solidity.expr.simps)
  qed
next
  case o: (NOT e)
  show ?case
  proof (cases "eupdate e")
    case (INT x11 x12)
```

```
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (UINT x21 x22)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (ADDRESS x3)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (BALANCE x4)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case THIS
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case SENDER
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case VALUE
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case t: TRUE
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case f: FALSE
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (LVAL x7)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case p: (PLUS x81 x82)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (MINUS x91 x92)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (EQUAL x101 x102)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (LESS x111 x112)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (AND x121 x122)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (OR x131 x132)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (NOT x131)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (CALL x181 x182)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case (ECALL x191 x192 x193)
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  next
    case CONTRACTS
    with o show ?thesis by (simp add:Statements.solidity.expr.simps)
  qed
next
  case (CALL x181 x182)
  show ?case by simp
next
  case (ECALL x191 x192 x193 x194)
```

```
  show ?case by simp
next
  case CONTRACTS
  show ?case by simp
qed

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.