Isabelle/Solidity

A shallow Embedding of Solidity in Isabelle/HOL

Diego Marmsoler and Asad Ahmed and Achim D. Brucker

May 8, 2026

Department of Computer Science, University of Exeter, Exeter, UK
{d.marmsoler, a.brucker, a.ahmed6}@exeter.ac.uk

Abstract

Smart contracts, typically written in high-level languages such as Solidity, are programs deployed on the blockchain to automate financial transactions. Due to the high-stakes nature of these applications, bugs can result in severe financial consequences. In this paper, we present a verification framework for Solidity smart contracts built within the Isabelle/HOL proof assistant. Our approach introduces a novel formalization of Soliditys storage model, a shallow embedding of its expressions and statements, and custom Isabelle commands to facilitate contract specification and verification. To aid in the verification process, we also provide a verification condition generator. We validate the semantics through a suite of unit tests and evaluate the framework using four case studies. The results demonstrate that our framework enables the verification of complex contracts with manageable effort. Furthermore, the use of shallow embedding significantly reduces verification complexity compared to a deep embedding approach. **Keywords:** Program Verification, Smart Contracts, Isabelle, Solidity

Contents

1 Introduction

This document presents a shallow embedding of Solidity in Isabelle. The embedding is described in a corresponding paper [Marmsoler et al.(2024)Marmsoler, Ahmed, and Brucker]. The shallow embedding is related to the corresponding deep embedding which is available as a separate AFP entry [Marmsoler and Brucker(2022)].

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).

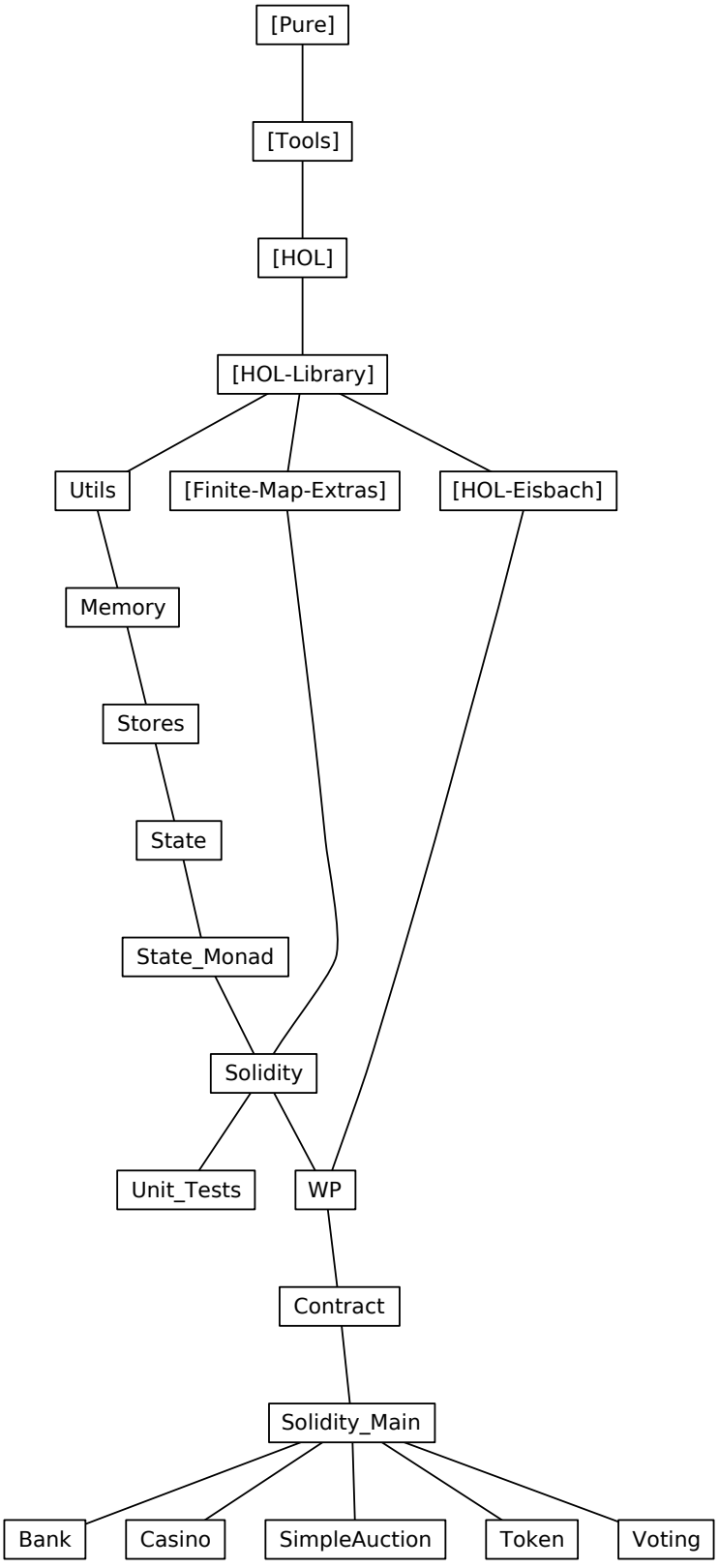

Figure 1.1: The Dependency Graph of the Isabelle Theories.

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.

theory *Utils*
imports
Main
"HOL-Library.Finite_Map"
"HOL-Library.Monad_Syntax"
begin

2.1 Relators (Utils)

lemma *set_listall:*
assumes *"⋀x. x ∈ set xs ⟹ (⋀y. R x y = (f x = y))"*
shows *"list_all2 R xs ys = (map f xs = ys)"*
⟨*proof*⟩

2.2 Fold (Utils)

lemma *fold_none_none[simp]:*
"fold (λx y. y ⤜ (λy'. f x y')) xs None = None"
⟨*proof*⟩

lemma *fold_take:*
assumes *"n < length xs"*
shows *"fold f (take (Suc n) xs) s = f (xs!n) (fold f (take n xs) s)"*
⟨*proof*⟩

lemma *fold_some_take_some:*
assumes *"fold (λx y. y ⤜ (λy'. f x y')) xs a = Some x"*
and *"n < length xs"*
obtains *y* **where** *"(fold (λx y. y ⤜ (λy'. f x y')) (take n xs) a) ⤜ (λy'. f (xs!n) y') = Some y"*
⟨*proof*⟩

lemma *fold_same:*
assumes *"∀x∈set xs. f x = g x"*
shows *"fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) xs L*
= fold (λx y. y ⤜ (λy'. g x ⤜ (λl. Some (l |∪| y')))) xs L"
⟨*proof*⟩

lemma *fold_f_none_none[simp]:*
assumes *"f a = None"*
shows *"fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) (a # xs) (Some X) = None"*
⟨*proof*⟩

lemma *fold_none_the_fold:*
assumes *"(fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) xs (Some X)) ≠ None*
∨ (fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) xs (Some (X |∪| Y))) ≠ None"
shows *"the (fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) xs (Some (X |∪| Y))) =*
(the (fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) xs (Some X))) |∪| Y"
⟨*proof*⟩

lemma *fold_some_some:*
shows *"fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) (take (n) xs) (Some (finsert x X))*
= fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) (take (n) xs) (Some X) ⤜ Some ∘ finsert x"
⟨*proof*⟩

lemma *fold_insert_same:*
assumes *"x ∉ fset (the (fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) (take n xs) (Some X)))"*
shows *"the (fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) (take n xs) (Some X)) = (the (fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) (take n xs) (Some (finsert x X)))) |-| {|x|}"*
⟨*proof*⟩

lemma *fold_some_diff:*
assumes *"x ∉ fset (the (fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) (take n xs) (Some {||})))"*
and *"(fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) (take n xs) (Some {||}) ≠ None"*
shows *"fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) (take n xs) (Some {||}) = Some ((the (fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) (take n xs) (Some {|x|}))) |-| {|x|})"*
⟨*proof*⟩

lemma *fold_none[simp]:*
"fold (λx y. y ⤜ g x) xs None = None"
⟨*proof*⟩

lemma *fold_some:*
assumes *"fold (λx y. y ⤜ (λy'. (f x) ⤜ (λl. Some (l |∪| y')))) xs X0 = Some X"*
shows *"∃X'. X0 = Some X' ∧ X' |⊆| X"*
⟨*proof*⟩

lemma *fold_some_ex:*
assumes *"fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) xs (Some X') = Some X"*
and *"i |∈| X"*
and *"i |∉| X'"*
shows *"∃x A. x ∈ set xs ∧ f x = Some A ∧ i |∈| A"*
⟨*proof*⟩

lemma *fold_some_subs:*
assumes *"fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) xs X' = Some X"*
and *"i ∈ set xs"*
shows *"∃A. f i = Some A ∧ A |⊆| X"*
⟨*proof*⟩

lemma *fold_subs_none:*
assumes *"fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) xs (Some X) = None"*
and *"X |⊆| Y"*
shows *"fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) xs (Some Y) = None"*
⟨*proof*⟩

lemma *fold_f_set_none:*
assumes *"a ∈ set xs"*
and *"f a = None"*
shows *"fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) xs (Some X) = None"*
⟨*proof*⟩

lemma *fold_f_set_some:*
assumes *"∀a ∈ set xs. f a ≠ None"*
shows *"fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) xs (Some X) ≠ None"*
⟨*proof*⟩

lemma *fold_disj:*
assumes *"∀x ∈ set xs. ∀L. f x = Some L ⟶ s |∩| L = {||}"*
and *"fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) xs (Some X) = Some L"*
and *"X |∩| s = {||}"*
shows *"s |∩| L = {||}"*
⟨*proof*⟩

lemma *fold_union_in:*
assumes *"fold (λx y. y ≫= (λy'. f x ≫= (λl. Some (l |∪| y')))) xs (Some L) = Some L'"*
and *"x |∈| L'"*
shows *"x |∈| L ∨ (∃n L''. n<length xs ∧ f (xs ! n) = Some L'' ∧ x |∈| L'')"*
⟨*proof*⟩

lemma *fold_subs:*
assumes *"∀x ∈ set xs. ∀L. f x = Some L ⟶ fset L ⊆ Y"*
and *"fold (λx y. y ≫= (λy'. f x ≫= (λl. Some (l |∪| y')))) xs (Some X) = Some L"*
and *"fset X ⊆ Y"*
shows *"fset L ⊆ Y"*
⟨*proof*⟩

lemma *fold_in_subs:*
assumes *"fold (λx y. y ≫= (λy'. f x ≫= (λl. Some (l |∪| y')))) xs (Some X) = Some L"*
and *"x ∈ set xs"*
and *"f x = Some S"*
shows *"S |⊆| L"*
⟨*proof*⟩

2.3 Take (Utils)

lemma *take_all:*
assumes *"∀n ≤ length xs. P (take n xs) (take n ys)"*
and *"length xs = length ys"*
shows *"P xs ys"*
⟨*proof*⟩

lemma *take_all1:*
assumes *"∀n ≤ length xs. P (take n xs)"*
shows *"P xs"*
⟨*proof*⟩

lemma *rev_induct2 [consumes 1, case_names Nil snoc]:*
assumes *"length xs = length ys"* **and** *"P [] []"*
and *"(⋀x xs y ys. length xs = length ys ⟹ P xs ys ⟹ P (xs @ [x]) (ys @ [y]))"*
shows *"P xs ys"*
⟨*proof*⟩

2.4 Filter (Utils)

lemma *length_filter_take_suc:*
assumes *"n<length daa"*
and *"P (daa!n)"*
shows *"length (filter P (take (Suc n) daa)) = Suc (length (filter P (take n daa)))"*
⟨*proof*⟩

2.5 Those (Utils)

lemma *those_map:*
assumes *"length xs = length ys"*
and *"those (map f (x # xs)) = Some (y # ys)"*
shows *"those (map f xs) = Some ys ∧ f x = Some y"*
⟨*proof*⟩

lemma *those_map_eq:*
assumes *"∀x ∈ set xs. ∀y. f x = Some y ⟶ g x = Some y"*
and *"∀x ∈ set xs. f x ≠ None"*
shows *"those (map f xs) = those (map g xs)"*
⟨*proof*⟩

lemma *those_map_none:*
assumes *"those (map f xs) = Some y"*
shows *"∀x ∈ set xs. f x ≠ None"*
⟨*proof*⟩

lemma *those_map_some[simp]:*
```
  "those (map Some xs) = Some xs"
```
⟨*proof*⟩

lemma *those_some_map:*
```
  assumes "those xs = Some xs'"
    shows "xs = map Some xs'"
```
⟨*proof*⟩

lemma *those_those:*
```
  assumes "those xs = Some xs'"
      and "those ys = Some ys'"
    shows "(those (xs @ ys)) = Some (xs' @ ys')"
```
⟨*proof*⟩

lemma *those_map_none_none:*
```
  assumes "those (map f as) = None"
  shows "∃x ∈ set as. f x = None"
```
⟨*proof*⟩

lemma *those_map_some_some:*
```
  assumes "∀x ∈ set as. f x ≠ None"
  shows "those (map f as) ≠ None"
```
⟨*proof*⟩

lemma *map_some_those_some:*
```
  assumes "length as = length ls"
      and "∀i<length as. map f ls ! i = Some (as ! i)"
    shows "those (map f ls) = Some as"
```
⟨*proof*⟩

2.6 Fold Map (Utils)

fun *fold_map* **where**
```
    "fold_map _ [] y = ([], y)"
  | "fold_map f (x # xs) y =
      (let (x', y') = f x y in
       (let (xs', y'') = fold_map f xs y'
      in (x' # xs', y'')))"
```

lemma *fold_map_length:*
```
  "length (fst (fold_map f ds m)) = length ds"
```
⟨*proof*⟩

lemma *fold_map_mono:*
```
  assumes "⋀x y x' y'. (f x y) = (x',y') ⟹ f' y' ≥ ((f' y):: nat)"
    and "fold_map f x y = (x', y')"
  shows "f' y' ≥ f' y"
```
⟨*proof*⟩

lemma *fold_map_geq:*
```
  assumes "⋀y x. f' (snd (f y x)) ≥ ((f' x):: nat)"
  shows "f' (snd (fold_map f x y)) ≥ f' y"
```
⟨*proof*⟩

lemma *fold_map_cong [fundef_cong]:*
```
  "a = b ⟹ xs = ys ⟹ (⋀x. x ∈ set xs ⟹ f x = g x)
    ⟹ fold_map f xs a = fold_map g ys b"
```
⟨*proof*⟩

lemma *fold_map_take_fst:*
```
  assumes "n < length (fst (fold_map f xs m))"
  shows "fst (fold_map f xs m) ! n = fst (f (xs ! n) (snd (fold_map f (take n xs) m)))"
```

⟨*proof*⟩

lemma *fold_map_take_snd:*
assumes *"n < length xs"*
shows *"snd (fold_map f (take (Suc n) xs) m) = snd (f (xs ! n) (snd (fold_map f (take n xs) m)))"*
⟨*proof*⟩

2.7 Prefix (Utils)

definition *prefix* **where** *"prefix m0 mf = (∃m'''. mf = m0@m''')"*

lemma *prefix_id[intro]: "prefix x x"* ⟨*proof*⟩

lemma *prefix_trans: "prefix x y ⟹ prefix y z ⟹ prefix x z"* ⟨*proof*⟩

definition *sprefix* **where** *"sprefix m0 mf = (∃m''' ≠ []. mf = m0@m''')"*

lemma *sprefix_append[simp]: "sprefix xs (xs@[y])"* ⟨*proof*⟩

lemma *sprefix_prefix: "sprefix m0 mf ⟹ prefix m0 mf"* ⟨*proof*⟩

lemma *sprefix_trans: "sprefix x y ⟹ sprefix y z ⟹ sprefix x z"* ⟨*proof*⟩

lemma *sprefix_length: "sprefix x y ⟹ length x < length y"* ⟨*proof*⟩

lemma *fold_map_prefix:*
assumes *"fold_map f ds m = (ls, m')"*
and *"⋀x y x' y'. f x y = (x', y') ⟹ prefix y y'"*
shows *"prefix m m'"*
⟨*proof*⟩

definition *length_append* **where** *"length_append m x = (length m, m@[x])"*

primrec *ofold :: "('a ⇒ 'b ⇒ 'b option) ⇒ 'a list ⇒ 'b ⇒ 'b option"* **where**
fold_Nil: "ofold f [] = Some" |
fold_Cons: "ofold f (x # xs) b = ofold f xs b ⤜ f x"

lemma *ofold_cong [fundef_cong]:*
"a = b ⟹ xs = ys ⟹ (⋀x. x ∈ set xs ⟹ f x = g x)
⟹ ofold f xs a = ofold g ys b"
⟨*proof*⟩

fun *K* **where** *"K f _ = f"*

fun *append* (**infixr** *"@@" 65*) **where** *"append xs x = xs @ [x]"*

abbreviation *case_list* **where** *"case_list l a b ≡ List.case_list a b l"*

2.8 Nth safe (Utils)

definition *nth_safe :: "'a list ⇒ nat ⇒ 'a option"* (**infixl** *"$" 100*)
where *"nth_safe xs i = (if i < length xs then Some (xs!i) else None)"*

lemma *nth_safe_some[simp]: "i < length xs ⟹ xs $ i = Some (xs!i)"* ⟨*proof*⟩
lemma *nth_safe_none[simp]: "i ≥ length xs ⟹ xs $ i = None"* ⟨*proof*⟩
lemma *nth_safe_length: "xs $ i = Some x ⟹ i < length xs"* ⟨*proof*⟩

definition *list_update_safe :: "'a list ⇒ nat ⇒ 'a ⇒ ('a list) option"*
where *"list_update_safe xs i a = (if i < length xs then Some (list_update xs i a) else None)"*

lemma *those_map_nth:*
assumes *"those (map f as) = Some xs"*
shows *"as $ x ⤜ f = xs $ x"*
⟨*proof*⟩

lemma *nth_in_set:*
  assumes *"xs $ x = Some y"*
  shows *"y ∈ set xs"*
  ⟨*proof*⟩

lemma *set_nth_some:*
  assumes *"y ∈ set xs"*
  shows *"∃x. xs $ x = Some y"*
  ⟨*proof*⟩

lemma *those_map_some_nth:*
  assumes *"those (map f as) = Some a"*
    and *"as $ x = Some y"*
  obtains *z* **where** *"f y = Some z"*
  ⟨*proof*⟩

lemma *nth_safe_prefix:*
  assumes *"m $ l = Some v"*
  and *"prefix m m'"*
shows *"m' $ l = Some v"*
  ⟨*proof*⟩

2.9 Some Basic Lemmas for Finite Maps (Utils)

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

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

2.10 Address class with example instantiation (Utils)

class *address = finite +*
  fixes *null :: 'a*

definition *aspace_carrier* **where** *"aspace_carrier={0::nat, 1, 2, 3, 4, 5, 6, 7, 8, 9}"*

lemma *aspace_carrier_finite: "finite aspace_carrier"* ⟨*proof*⟩

typedef *aspace = aspace_carrier*
  ⟨*proof*⟩

setup_lifting *type_definition_aspace*

lift_definition *A0 :: aspace* **is** *0* ⟨*proof*⟩
lift_definition *A1 :: aspace* **is** *1* ⟨*proof*⟩
lift_definition *A2 :: aspace* **is** *2* ⟨*proof*⟩
lift_definition *A3 :: aspace* **is** *3* ⟨*proof*⟩
lift_definition *A4 :: aspace* **is** *4* ⟨*proof*⟩
lift_definition *A5 :: aspace* **is** *5* ⟨*proof*⟩
lift_definition *A6 :: aspace* **is** *6* ⟨*proof*⟩
lift_definition *A7 :: aspace* **is** *7* ⟨*proof*⟩
lift_definition *A8 :: aspace* **is** *8* ⟨*proof*⟩
lift_definition *A9 :: aspace* **is** *9* ⟨*proof*⟩

lemma *aspace_finite: "finite (UNIV::aspace set)"*
  ⟨*proof*⟩

lemma *A1_neq_A0[simp]: "A1 ≠ A0"*
  ⟨*proof*⟩

instantiation *aspace :: address*
begin
  definition *null_def: "null = A0"*
  instance ⟨*proof*⟩
end

instantiation *aspace :: equal*
begin

definition *"HOL.equal (x::aspace) y ⟷ Rep_aspace x = Rep_aspace y"*

instance ⟨*proof*⟩

end

2.11 Common lemmas for sums (Utils)

lemma *sum_addr:*
  fixes *f::"('a::address)⇒nat"*
  shows *"(∑ ad∈UNIV. f ad) = (∑ ad|ad ≠ addr. f ad) + f addr"*
⟨*proof*⟩

lemma *sum_addr2:*
  fixes *f::"'a::address⇒nat"*
  assumes *"addr ∈ A"*
  shows *"(∑ ad∈A. f ad) = (∑ ad|ad∈A ∧ ad ≠ addr. f ad) + f addr"*
⟨*proof*⟩

lemma *sum_addr3:*
  fixes *f::"'a::address⇒nat"*
  assumes *"addr ∉ A"*
  shows *"(∑ ad∈A. f ad) = (∑ ad|ad∈A ∧ ad ≠ addr. f ad)"*
⟨*proof*⟩

2.12 Pred Some (Utils)

definition *pred_some* **where**
  "pred_some P v = (∃ v'. v = Some v' ∧ P v')"

definition *fs_disj_fs* **where**
  "fs_disj_fs B C = pred_some (λC'. pred_some (λB'. C' |∩| B' = {||}) B) C"

definition *s_disj_fs* **where**
  "s_disj_fs B C = pred_some (λC'. fset C' ∩ B = {}) C"

definition *s_eq_fs* **where**
  "s_eq_fs B C = pred_some (λC'. B = fset C') C"

definition *s_subs_fs* **where**
  "s_subs_fs B C = pred_some (λC'. B ⊆ fset C') C"

definition *fs_subs_fs* **where**
  "fs_subs_fs B C = pred_some (λB'. B' |⊆| C) B"

definition *fs_subs_s* **where**
  "fs_subs_s B C = pred_some (λB'. fset B' ⊆ C) B"

definition *s_union_fs* **where**
  "s_union_fs A B C = pred_some (λC'. A = B ∪ fset C') C"

definition *loc* **where**
  "loc m = {l. l < length m}"

lemma *s_disj_fs_prefix:*

assumes *"prefix m m'"*
and *"s_disj_fs (loc m') X"*
shows *"s_disj_fs (loc m) X"*
⟨*proof*⟩

lemma *s_union_fs_s_union_fs_union:*
assumes *"s_union_fs B X B'"*
and *"A = (B - C) ∪ D"*
and *"A' = Some ((the B' |-| C') |∪| the D')"*
and *"C ∩ X = {}"*
and *"C = fset C'"*
and *"D = fset (the D')"*
shows *"s_union_fs A X A'"*
⟨*proof*⟩

lemma *s_union_fs_diff:*
assumes *"s_union_fs A B C"*
and *"B ∩ fset (the C) = {}"*
shows *"(A - B) = fset (the C)"*
⟨*proof*⟩

lemma *s_disj_fs_loc_fold:*
assumes *"s_disj_fs (loc m0) (fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) xs (X))"*
and *"s_disj_fs (loc m0) X"*
shows *"s_disj_fs (loc m0) (fold (λx y. y ⤜ (λy'. f x ⤜ (λl. Some (l |∪| y')))) (take n xs) (X))"*
⟨*proof*⟩

lemma *s_disj_union_fs: "s_disj_fs B C ⟹ s_union_fs A B C ⟹ fset (the C) = A - B"*
⟨*proof*⟩

lemma *disj_empty[simp]: "s_disj_fs X (Some {||})"* ⟨*proof*⟩

lemma *s_union_fs_s_union_fs_diff:*
assumes *"s_union_fs X Y Z"*
and *"X' = X - {a}"*
and *"the Z' = the Z |-| {|a|}"*
and *"Z' ≠ None"*
and *"a ∉ Y"*
shows *"s_union_fs X' Y Z'"*
⟨*proof*⟩

2.13 Termination Lemmas (Utils)

lemma *card_less_card:*
assumes *"m $ p1 = Some a"*
and *"p1 |∉| s1"*
shows *"card ({0..length m} - insert p1 (fset s1)) < card ({0..length m} - fset s1)"*
⟨*proof*⟩

end
theory *State_Monad*
imports *State "HOL-Library.Monad_Syntax" Utils*
begin

2.14 state Monad with Exceptions (State_Monad)

datatype *('n, 'e) result =*
Normal (normal: 'n)
| Exception (ex: 'e)
| NT

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

```
        | (e) e s where "x = Exception (e, s)"
        | (t) "x = NT"
```
⟨proof⟩

```
typedef ('a, 'e, 's) state_monad = "UNIV::('s ⇒ ('a × 's, 'e × 's) result) set"
  morphisms execute create
```
⟨proof⟩

```
named_theorems execute_simps "simplification rules for execute"

lemma execute_Let [execute_simps]:
  "execute (let x = t in f x) = (let x = t in execute (f x))"
```
⟨proof⟩

2.14.1 Code Generator Setup

```
code_datatype create

lemma execute_create[execute_simps, code]: "execute (create f) = f"
```
⟨proof⟩

```
declare execute_inverse[simp]

lemma execute_ext[intro]: "(⋀x. (execute m1 x = execute m2 x)) ⟹ m1 = m2"
```
⟨proof⟩

2.14.2 Fundamental Definitions

```
definition return :: "'a ⇒ ('a, 'e, 's) state_monad"
  where "return a = create (λs. Normal (a, s))"

lemma execute_return [execute_simps]:
  "execute (return x) = Normal ∘ Pair x"
```
⟨proof⟩

```
lemma execute_returnE:
  assumes "execute (return x) s = Normal (a, s')"
  shows "x = a" and "s = s'"
```
⟨proof⟩

```
definition throw :: "'e ⇒ ('a, 'e, 's) state_monad"
  where "throw e = create (λs. Exception (e, s))"

lemma execute_throw [execute_simps]:
  "execute (throw x) s = Exception (x, s)"
```
⟨proof⟩

```
definition bind :: "('a, 'e, 's) state_monad ⇒ ('a ⇒ ('b, 'e, 's) state_monad) ⇒ ('b, 'e, 's)
state_monad" (infixl ">>=" 60)
where "bind f g = create (λs. (case (execute f) s of
                      Normal (a, s') ⇒ execute (g a) s'
                    | Exception e ⇒ Exception e
                    | NT ⇒ NT))"

adhoc_overloading Monad_Syntax.bind ⇌ bind

lemma execute_bind [execute_simps]:
  "execute f s = Normal (x, s') ⟹ execute (f ⤜ g) s = execute (g x) s'"
  "execute f s = Exception e ⟹ execute (f ⤜ g) s = Exception e"
  "execute f s = NT ⟹ execute (f ⤜ g) s = NT"
```
⟨proof⟩

```
lemma execute_bind_normal_E:
  assumes "execute (f ⤜ g) s = Normal (a, s')"
  obtains (1) s'' x where "execute f s = Normal (x, s'')" and "execute (g x) s'' = Normal (a, s')"
```
⟨proof⟩

lemma *execute_bind_exception_E:*
assumes *"execute (f ≫= g) s = Exception (x, s')"*
obtains *(1) "execute f s = Exception (x, s')"*
| (2) a s'' where "execute f s = Normal (a,s'')" and "execute (g a) s'' = Exception (x,s')"
⟨*proof*⟩

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

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

2.14.3 The Monad Laws

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

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

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

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

(≫=) is associative

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

2.14.4 Basic Conguruence Rules

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

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

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

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

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

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

2.14.5 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.

definition *get :: "('s, 'e, 's) state_monad"* **where**
"get = create (λs. Normal (s, s))"

lemma *execute_get [execute_simps]:*
"execute get = (λs. Normal (s, s))"
⟨*proof*⟩

definition *put :: "'s ⇒ (unit, 'e, 's) state_monad"* **where**
"put s = create (K (Normal ((), s)))"

lemma *execute_put [execute_simps]:*
"execute (put s) = K (Normal ((), s))"
⟨*proof*⟩

definition *update :: "('s ⇒ 'a × 's) ⇒ ('a, 'e, 's) state_monad"* **where**
"update f = create (λs. Normal (f s))"

lemma *execute_update [execute_simps]:*
"execute (update f) = (λs. Normal (f s))"
⟨*proof*⟩

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

definition *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.

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

lemma *execute_modify [execute_simps]:*
"execute (modify f) s = Normal ((), f s)"
⟨*proof*⟩

primrec *mfold :: "('a,'e,'s) state_monad ⇒ nat ⇒ ('a list,'e,'s) state_monad"*
where
"mfold m 0 = return []"
| "mfold m (Suc n) =
do {
l ← m;
ls ← mfold m n;
return (l # ls)
}"

2.14.6 Some basic examples

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

lemma *"do {*

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

2.14.7 Conditional Monad

fun *cond_monad:: "('s ⇒ bool) ⇒ ('a, 'e, 's) state_monad ⇒ ('a, 'e, 's) state_monad ⇒ ('a, 'e, 's) state_monad"* **where**
```
"cond_monad c mt mf =
  do {
    s ← get;
    if (c s) then mt else mf
  }"
```

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

lemma *execute_option [execute_simps]:*
```
  "⋀y. f s = Some y ⟹ execute (option e f) s = execute (return y) s"
  "f s = None ⟹ execute (option e f) s = execute (throw e) s"
```
⟨proof⟩

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

lemma *execute_assert [execute_simps]:*
```
  "t s ⟹ execute (assert e t) s = execute (return ()) s"
  "¬ t s ⟹ execute (assert e t) s = execute (throw e) s"
```
⟨proof⟩

2.14.8 Setup for Partial Function Package

We can make result into a pointed cpo:

- The order is obtained by combinin function order with result order
- The least element is NT

definition *effect :: "('a, 'b, 'c) state_monad ⇒ 'c ⇒ 'a × 'c + 'b × 'c ⇒ bool"* **where**
```
  effect_def: "effect c h r ⟷ is_Normal (execute c h) ∧ r = Inl (normal (execute c h)) ∨
is_Exception (execute c h) ∧ r = Inr (ex (execute c h))"
```

lemma *effectE:*
```
  assumes "effect c h r"
  obtains (normal) "is_Normal (execute c h) ∧ r = Inl (normal (execute c h))"
  | (exception) "is_Exception (execute c h) ∧ r = Inr (ex (execute c h))"
```
⟨proof⟩

abbreviation *"empty_result ≡ create (λs. NT)"*
abbreviation *"result_ord ≡ flat_ord NT"*
abbreviation *"result_lub ≡ flat_lub NT"*

definition *sm_ord :: "('a, 'e, 's) state_monad ⇒ ('a, 'e, 's) state_monad ⇒ bool"* **where**
```
  "sm_ord = img_ord execute (fun_ord result_ord)"
```

definition *sm_lub :: "('a, 'e, 's) state_monad set ⇒ ('a, 'e, 's) state_monad"* **where**
```
  "sm_lub = img_lub execute create (fun_lub result_lub)"
```

lemma *sm_lub_empty: "sm_lub {} = empty_result"*
⟨proof⟩

lemma *sm_ordI:*
assumes *"⋀h. execute x h = NT ∨ execute x h = execute y h"*
shows *"sm_ord x y"*
⟨*proof*⟩

lemma *sm_ordE:*
assumes *"sm_ord x y"*
obtains *"execute x h = NT" | "execute x h = execute y h"*
⟨*proof*⟩

lemma *sm_interpretation: "partial_function_definitions sm_ord sm_lub"*
⟨*proof*⟩

interpretation *sm: partial_function_definitions sm_ord sm_lub*
rewrites *"sm_lub {} ≡ empty_result"*
⟨*proof*⟩

named_theorems *mono*

declare *sm.const_mono[mono]*
declare *Partial_Function.call_mono[mono]*

The success predicate requires a state monad sm starting in state s to terminate successfully in state s' with return value a.

definition *success :: "('a, 'e, 's) state_monad ⇒ 's ⇒ 's ⇒ 'a ⇒ bool"* **where**
success_def: "success sm s s' a ⟷ execute sm s ≠ NT"

We can show that every predicate P is admissible if we assume successful termination.

lemma *sm_step_admissible:*
"ccpo.admissible (fun_lub result_lub) (fun_ord result_ord) (λxa. ∀h r. is_Normal (xa h) ∧ r = Inl (normal (xa h)) ∨ is_Exception (xa h) ∧ r = Inr (ex (xa h)) ⟶ P h r)"
⟨*proof*⟩

lemma *admissible_sm:*
"sm.admissible (λf. ∀x h r. effect (f x) h r ⟶ P x h r)"
⟨*proof*⟩

Now we can derive an induction rule for proving partial correctness properties. Note that this rule requires successful termination.

lemma *fixp_induct_sm:*
fixes *F :: "'c ⇒ 'c"* **and**
U :: "'c ⇒ 'b ⇒ ('a, 'e, 's) state_monad" **and**
C :: "('b ⇒ ('a, 'e, 's) state_monad) ⇒ 'c" **and**
P :: "'b ⇒ 's ⇒ 'a × 's + 'e × 's ⇒ bool"
assumes *mono: "⋀x. monotone (fun_ord sm_ord) sm_ord (λf. U (F (C f)) x)"*
assumes *eq: "f ≡ C (ccpo.fixp (fun_lub sm_lub) (fun_ord sm_ord) (λf. U (F (C f))))"*
assumes *inverse2: "⋀f. U (C f) = f"*
assumes *step: "⋀f x h r. (⋀x h r. effect (U f x) h r ⟹ P x h r) ⟹ effect (U (F f) x) h r ⟹ P x h r"*
assumes *defined: "effect (U f x) h r"*
shows *"P x h r"*
⟨*proof*⟩

We now need to setup the new sm mode for the partial function package.

⟨*ML*⟩

2.14.9 Monotonicity Results

abbreviation *"mono_sm ≡ monotone (fun_ord sm_ord) sm_ord"*

lemma *execute_bind_case:*
"execute (f ⤜ g) h = (case (execute f h) of

```
    Normal (x, h') ⇒ execute (g x) h' | Exception e ⇒ Exception e | NT ⇒ NT)"
```
⟨*proof*⟩

lemma *bind_mono [partial_function_mono,mono]:*
assumes *mf: "mono_sm B"* **and** *mg: "⋀y. mono_sm (λf. C y f)"*
shows *"mono_sm (λf. B f ⤜ (λy. C y f))"*
⟨*proof*⟩

lemma *throw_monad_mono[mono]: "mono_sm (λ_. throw e)"*
⟨*proof*⟩

lemma *return_monad_mono[mono]: "mono_sm (λ_. return x)"*
⟨*proof*⟩

lemma *option_monad_mono[mono]: "mono_sm (λ_. option E x)"*
⟨*proof*⟩

definition *exc:: "('a, 'b, 'c) state_monad ⇒ ('a, 'b, 'c) state_monad"*
where *"exc m ≡ create (λs. case execute m s of Normal (v,s') ⇒ Normal (v, s')*
| Exception (e, s') ⇒ Exception (e, s)
| NT ⇒ NT)"

lemma *exc_mono[mono]:*
fixes *m::"('b ⇒ ('c, 'e, 'f) state_monad) ⇒ ('x, 'y, 'z) state_monad"*
assumes *mf: "mono_sm (λcall. (m call))"*
shows *"mono_sm (λcall. (exc (m call)))"*

⟨*proof*⟩

end

3 Stores and State

In this chapter, we formalize the different types of stores for Solidity. In particular, we present a formalization for Storage, Memory, Calldata, and Stack. Then we define the state consisting of a configuration of all the stores as well as account balances.

```
theory Memory
  imports Utils
begin

class vtype =
  fixes to_nat :: "'a ⇒ nat option"
```

3.1 Memory (Memory)

```
type_synonym location = nat

datatype 'v mdata =
  is_Value: Value (vt: 'v)
| is_Array: Array (ar: "location list")

definition case_memory where
  "case_memory m l vf af ≡
    (case m$l of
      Some (mdata.Value v) ⇒ vf v
    | Some (mdata.Array xs) ⇒ af xs
    | None ⇒ None)"

lemma case_memory_cong[fundef_cong]:
  assumes "⋀v. m$l = Some (mdata.Value v) ⟹ vf1 v = vf2 v"
      and "⋀xs. m$l = Some (mdata.Array xs) ⟹ af1 xs = af2 xs"
    shows "case_memory m l vf1 af1 = case_memory m l vf2 af2"
  ⟨proof⟩

type_synonym 'v memory = "'v mdata list"
```

3.2 Array Lookup (Memory)

```
fun marray_lookup ::
    "'v::vtype memory ⇒ 'v list ⇒ location ⇒ (location × location list × nat) option"
  where
  "marray_lookup _ [] _  = None"
| "marray_lookup m [i] l =
    case_memory m l
      (K None)
      (λxs. to_nat i ⤜ (λi. Some (l, xs, i)))"
| "marray_lookup m (i # is) l =
    case_memory m l
      (K None)
      (λxs. to_nat i ⤜ ($) xs ⤜ marray_lookup m is)"

lemma marray_lookup_obtain_single:
  assumes "marray_lookup m [i] l = Some a"
  obtains xs i''
  where "m $ l = Some (mdata.Array xs)"
    and "to_nat i = Some i''"
    and "a = (l, xs, i'')"
  ⟨proof⟩
```

lemma *marray_lookup_obtain_multi:*
  assumes *"marray_lookup m (i # i' # is) l = Some a"*
  obtains *xs i'' l'*
  where *"m $ l = Some (mdata.Array xs)"*
    and *"to_nat i = Some i''"*
    and *"xs $ i'' = Some l'"*
    and *"marray_lookup m (i'#is) l' = Some a"*
  ⟨*proof*⟩

lemma *marray_lookup_prefix:*
  assumes *"marray_lookup m xs l = Some x"*
      and *"prefix m m'"*
    shows *"marray_lookup m' xs l = Some x"*
  ⟨*proof*⟩

lemma *marray_lookup_prefix_some:*
  assumes *"xs ≠ []"*
    and *"marray_lookup m (xs@ys) l = Some y"*
  shows *"∃y. marray_lookup m xs l = Some y"*
  ⟨*proof*⟩

lemma *marray_lookup_append:*
  assumes *"xs ≠ []"*
      and *"ys ≠ []"*
    shows *"marray_lookup m (xs @ ys) l*
        = marray_lookup m xs l ≫= (λ(l', ls, i). (ls $ i) ≫= (λi. marray_lookup m ys i))"
  ⟨*proof*⟩

3.3 Memory Lookup (Memory)

fun *mlookup :: "'v::vtype memory ⇒ 'v list ⇒ location ⇒ location option"* **where**
  "mlookup m [] l = Some l"
| "mlookup m xs l = marray_lookup m xs l ≫= (λ(_, xs', i). xs' $ i)"

lemma *mlookup_obtain_empty:*
  assumes *"mlookup m [] l = Some a"*
  shows *"a = l"*
  ⟨*proof*⟩

lemma *mlookup_obtain_single:*
  assumes *"mlookup m [i] l = Some a"*
  obtains *xs i'*
  where *"m $ l = Some (mdata.Array xs)"*
    and *"to_nat i = Some i'"*
    and *"xs$i' = Some a"*
  ⟨*proof*⟩

lemma *mlookup_obtain_nempty1:*
  assumes *"mlookup m (x#xs) l = Some aa"*
  obtains *a xs' i i'*
  where *"marray_lookup m (x#xs) l = Some (a, xs', i)"*
    and *"xs' $ i = Some i'"*
    and *"aa = i'"*
  ⟨*proof*⟩

lemma *mlookup_obtain_nempty2:*
  assumes *"mlookup m (i # is) l = Some l'"*
  obtains *ls i' l''*
  where *"m$l = Some (mdata.Array ls)"*
    and *"to_nat i = Some i'"*
    and *"ls $ i' = Some l''"*
    and *"mlookup m is l'' = Some l'"*
  ⟨*proof*⟩

lemma *mlookup_start_some:*
assumes *"mlookup m (iv#is) l = Some l'"*
shows *"∃x. m$l = Some x"*
⟨*proof*⟩

lemma *mlookup_same:*
assumes *"xs ≠ []"*
and *"m $ l1 = m $ l2"*
shows *"mlookup m xs l1 = mlookup m xs l2"*
⟨*proof*⟩

lemma *mlookup_prefix_mlookup:*
assumes *"mlookup m xs0 l = Some x0"*
and *"prefix m m'"*
shows *"mlookup m' xs0 l = Some x0"*
⟨*proof*⟩

lemma *mlookup_append:*
"mlookup m (xs @ ys) l = mlookup m xs l ⤜ mlookup m ys"
⟨*proof*⟩

3.4 Memory Update (Memory)

fun *mupdate :: "'v::vtype list ⇒ location × 'v mdata × 'v memory ⇒ 'v memory option"* **where**
"mupdate xs (l, v, m) = mlookup m xs l ⤜ (λl. list_update_safe m l v)"

lemma *mvalue_update_obtain:*
assumes *"mupdate xs (l,v,m) = Some x"*
obtains *l'*
where *"mlookup m xs l = Some l'"*
and *"l' < length m"*
and *"x = m[l':=v]"*
⟨*proof*⟩

lemma *mvalue_update_length:*
assumes *"mupdate is (ml, v, m) = Some m'"*
shows *"length m' = length m"*
⟨*proof*⟩

3.5 Memory Locations (Memory)

fun *locations :: "'v mdata list ⇒ 'v::vtype list ⇒ nat ⇒ nat fset option"* **where**
"locations m [] l = Some ({|||})"
| "locations m (i#is) l =
case_memory m l
(K None)
(λxs. to_nat i ⤜ ($) xs ⤜ (λl'. locations m is l' ⤜ (λls. Some (finsert l ls))))"

lemma *locations_obtain:*
assumes *"locations m (i # is) l = Some L"*
obtains *as i' l' L'*
where *"m$l = Some (mdata.Array as)"*
and *"to_nat i = Some i'"*
and *"as $ i' = Some l'"*
and *"locations m is l' = Some L'"*
and *"L = (finsert l L')"*
⟨*proof*⟩

lemma *locations_l_in_L:*
assumes *"locations m (i#is') l = Some L"*
shows *"l |∈| L"*
⟨*proof*⟩

lemma *locations_same:*
assumes *"locations m xs0 l = Some L"*

and *"∀l|∈|L. m' $ l = m $ l"*
shows *"locations m' xs0 l = Some L"*
⟨*proof*⟩

lemma *locations_append_subset:*
assumes *"locations m (xs @ xs') l = Some L"*
obtains *L'*
where *"locations m xs l = Some L'"*
and *"L' |⊆| L"*
⟨*proof*⟩

lemma *locations_prefix_locations:*
assumes *"locations m xs0 l = Some L"*
and *"prefix m m'"*
shows *"locations m' xs0 l = Some L"*
⟨*proof*⟩

lemma *locations_subs_loc:*
assumes *"locations m xs0 l = Some L"*
shows *"fset L ⊆ loc m"*
⟨*proof*⟩

3.6 Locations and Array Lookup (Memory)

lemma *locations_marray_lookup_same:*
assumes *"locations m1 is l = Some L"*
and *"⋀l. l |∈| L ⟹ m1 $ l = m2 $ l"*
shows *"marray_lookup m1 is l = marray_lookup m2 is l"*
⟨*proof*⟩

lemma *marray_lookup_in_locations:*
assumes *"marray_lookup m is l = Some (l'', xs, i)"*
and *"locations m is l = Some L"*
shows *"l'' |∈| L"*
⟨*proof*⟩

lemma *marray_lookup_update_same:*
assumes *"locations m xs l = Some L"*
and *"¬ (l' |∈| L)"*
shows *"marray_lookup m xs l = marray_lookup (m[l':=v']) xs l"*
⟨*proof*⟩

lemma *marray_lookup_locations_some:*
assumes *"marray_lookup m xs l = Some (l0, xs', i)"*
and *"xs' $ i = Some i'"*
shows *"∃L. locations m xs l = Some L"*
⟨*proof*⟩

lemma *marray_lookup_in_locations2:*
assumes *"xs ≠ []"*
and *"ys ≠ []"*
and *"marray_lookup m xs l = Some (l0, xs0, i0)"*
and *"xs0 $ i0 = Some ll"*
and *"locations m (xs@ys) l = Some L"*
shows *"ll |∈| L"*
⟨*proof*⟩

3.7 Locations and Lookup (Memory)

lemma *mlookup_update_val:*
assumes *"mlookup m xs l = Some l'"*
and *"locations m xs l = Some L"*
and *"¬ (l'' |∈| L)"*
shows *"mlookup (m[l'':=v]) xs l = Some l'"*

⟨*proof*⟩

lemma *mlookup_locations_some:*
assumes *"mlookup m xs0 l = Some l'"*
shows *"∃L. locations m xs0 l = Some L"*
⟨*proof*⟩

lemma *mlookup_update_same_nempty:*
assumes *"mlookup m (x#xs) l1 = Some l1'"*
and *"locations m (x#xs) l1 = Some L"*
and *"¬ (l2 |∈| L)"*
shows *"mlookup (m[l2:=v']) (x#xs) l1 = mlookup m (x#xs) l1"*
⟨*proof*⟩

lemma *mlookup_in_locations:*
assumes *"ys ≠ []"*
and *"mlookup m xs l = Some l'"*
and *"locations m (xs@ys) l = Some L"*
shows *"l' |∈| L"*
⟨*proof*⟩

lemma *mlookup_access_same:*
assumes *"locations m1 xs l = Some L"*
and *"mlookup m1 xs l = Some l'"*
and *"⋀l. l |∈| L ⟹ m1 $ l = m2 $ l"*
and *"m1 $ l' = m2 $ l'"*
shows *"mlookup m2 xs l ≫= ($) m2 = m1 $ l'"*
⟨*proof*⟩

lemma *mlookup_same_locations:*
assumes *"mlookup m1 xs l = Some l'"*
and *"locations m1 xs l = Some L"*
and *"∀l |∈| L. m1 $ l = m2 $ l"*
shows *"mlookup m2 xs l = Some l'"*
⟨*proof*⟩

lemma *mlookup_append_same:*
assumes *"ys ≠ []"*
and *"mlookup m xs1 l1 = Some l1'"*
and *"m $ l1' = Some l1''"*
and *"mlookup m xs2 l2 ≫= ($) m = Some l1''"*
shows *"mlookup m (xs1 @ ys) l1 = mlookup m (xs2 @ ys) l2"* (**is** *"?LHS=?RHS"*)
⟨*proof*⟩

lemma *locations_union_nth:*
assumes *"xs = x#xs'"*
and *"m0 $ l0 = m1 $ l1"*
and *"mlookup m0 [x] l0 = Some l"*
and *"locations m0 xs' l = Some L"*
and *"∀l |∈| L. m0$l=m1$l"*
shows *"locations m1 xs l1 = Some (finsert l1 L)"*
⟨*proof*⟩

lemma *locations_union_mlookup_nth:*
assumes *"ys = y#ys'"*
and *"mlookup m0 xs l = Some l'"*
and *"m0 $ l'' = m1 $ l'"*
and *"mlookup m0 [y] l'' = Some l1"*
and *"locations m0 xs l = Some L0"*
and *"∀l |∈| L0. m0$l=m1$l"*
and *"locations m0 ys' l1 = Some L1"*
and *"∀l |∈| L1. m0$l=m1$l"*
shows *"locations m1 (xs @ ys) l = Some (finsert l' L0 |∪| L1)"*
⟨*proof*⟩

lemma *locations_union_mlookup:*
  assumes *"mlookup m xs l = Some l'"*
    and *"locations m xs l = Some L0"*
    and *"locations m ys l' = Some L1"*
  shows *"locations m (xs @ ys) l = Some (L0 |∪| L1)"*
⟨*proof*⟩

lemma *mlookup_locations_subs:*
  assumes *"mlookup m xs l = Some l'"*
    and *"locations m (xs @ ys) l = Some L0"*
    and *"locations m ys l' = Some L1"*
  shows *"L1 |⊆| L0"*
⟨*proof*⟩

proposition *is_none_mlookup_locations:*
  assumes *"¬ Option.is_none (mlookup m xs l)"*
  shows *"¬ Option.is_none (locations m xs l)"*
⟨*proof*⟩

lemma *locations_app_mlookup_exists:*
  assumes *"locations m (xs @ ys) l0 = Some L"*
    and *"mlookup m xs l0 = Some l1"*
  shows *"∃L' L''. locations m xs l0 = Some L' ∧ locations m ys l1 = Some L'' ∧ L = L' |∪| L''"*
⟨*proof*⟩

lemma *locations_cons_mlookup_exists:*
  assumes *"locations m0 (z#zs) l0 = Some L"*
    and *"mlookup m0 [z] l0 = Some l1"*
  shows *"∃L'. locations m0 zs l1 = Some L' ∧ L' |⊆| L"*
⟨*proof*⟩

lemma *mlookup_mlookup_mlookup:*
  assumes *"mlookup m0 ys l1 = Some l1'"*
    and *"m1 $ l1' = m0 $ l0'"*
    and *"zs ≠ []"*
    and *"∀l|∈|the (locations m0 zs l0'). m0 $ l = m1 $ l"*
    and *"∀l|∈|the (locations m0 ys l1). m0 $ l = m1 $ l"*
    and *"mlookup m0 zs l0' = Some l2"*
  shows *"mlookup m1 (ys @ zs) l1 = Some l2"*
⟨*proof*⟩

locale *data =*
  fixes *Value :: "'v::vtype ⇒ 'd"*
    and *Array :: "'d list ⇒ 'd"*
begin

3.8 Memory Locations (Memory)

function *range_safe :: "location fset ⇒ 'v memory ⇒ location ⇒ (location fset) option"* **where**
  "range_safe s m l =
   (if l |∈| s then None else
      case_memory m l
        (λv. Some {|l|})
        (λxs. fold
          (λx y. y ≫= (λy'. (range_safe (finsert l s) m x) ≫= (λl. Some (l |∪| y'))))
          xs
          (Some {|l|})))"
⟨*proof*⟩
termination
  ⟨*proof*⟩

lemma *range_safe_obtains:*
  assumes *"range_safe s m l = Some x"*

```
    obtains
    (1) v where "l |∉| s"
    and "m $ l = Some (mdata.Value v)"
    and "x = {|l|}"
  | (2) xs where "l |∉| s"
    and "m $ l = Some (mdata.Array xs)"
    and "fold
          (λx y. y ⤜ (λy'. (range_safe (finsert l s) m x) ⤜ (λl. Some (l |∪| y'))))
          xs
          (Some {|l|})
        = Some x"
```
⟨*proof*⟩

lemma *range_safe_subs:*
```
  assumes "range_safe s m l = Some X"
  shows "l |∈| X"
```
⟨*proof*⟩

lemma *range_safe_subs2:*
```
  assumes "range_safe s m l = Some X"
    shows "fset X ⊆ loc m"
```
⟨*proof*⟩

lemma *range_safe_obtains_subset:*
```
  assumes "range_safe s m l = Some L"
      and "m $ l = Some (mdata.Array xs)"
      and "l' ∈ set xs"
    obtains L' where "range_safe (finsert l s) m l' = Some L'" and "L' |⊆| L"
```
⟨*proof*⟩

lemma *range_safe_nin_same:*
```
  assumes "range_safe s m l = Some L"
      and "∀l |∈| s' - s. l |∉| L"
    shows "range_safe s' m l = Some L"
```
⟨*proof*⟩

lemma *range_safe_same:*
```
  assumes "range_safe s m l = Some L"
      and "∀l'|∈|L. m' $ l' = m $ l'"
    shows "range_safe s m' l = Some L"
```
⟨*proof*⟩

lemma *range_safe_same4:*
```
  assumes "range_safe s m l = Some L"
      and "∀l'|∈|L. (∃xs. m' $ l' = Some (mdata.Array xs) ∧ m $ l' = Some (mdata.Array xs)) ∨ (∃xs.
m' $ l' = Some (mdata.Value xs))"
    shows "∃L'. range_safe s m' l = Some L'"
```
⟨*proof*⟩

lemma *range_safe_subs3:*
```
  assumes "range_safe s m l = Some L"
      and "∀l'|∈|L. (∃xs. m' $ l' = Some (mdata.Array xs) ∧ m $ l' = Some (mdata.Array xs)) ∨ (∃xs.
m' $ l' = Some (mdata.Value xs))"
    shows "∃L'. range_safe s m' l = Some L' ∧ L' |⊆| L"
```
⟨*proof*⟩

lemma *range_safe_subset_same:*
```
  assumes "range_safe s m l = Some x"
    and "s' |⊆| s"
  shows "range_safe s' m l = Some x"
```
⟨*proof*⟩

lemma *range_safe_in_subs:*
```
  assumes "range_safe s m l = Some L"
```

and *"l' |∈| L"*
shows *"∃L'. range_safe s m l' = Some L' ∧ L' |⊆| L"*
⟨*proof*⟩

lemma *range_safe_disj:*
"∀L. range_safe s m l = Some L ⟶ s |∩| L = {||}"(**is** *"?P s m l"*)
⟨*proof*⟩

lemma *range_range:*
assumes *"range_safe s0 m l1 = Some L1"*
and *"range_safe s1 m l1 = Some L2"*
shows *"L1 = L2"*
⟨*proof*⟩

lemma *range_safe_prefix:*
assumes *"prefix m m'"*
and *"range_safe s m l = Some L"*
shows *"range_safe s m' l = Some L"*
⟨*proof*⟩

lemma *range_safe_locations:*
assumes *"range_safe s m l = Some L"*
and *"locations m xs l = Some L'"*
shows *"L' |⊆| L"*
⟨*proof*⟩

lemma *range_safe_l_in_L:*
assumes *"range_safe s m l = Some L"*
and *"x |∈| L"*
and *"m $ x = Some (mdata.Array xs)"*
and *"l' ∈ set xs"*
shows *"l' |∈| L"*
⟨*proof*⟩

lemma *range_safe_marray_lookup:*
assumes *"xs ≠ []"*
and *"range_safe s m l = Some L"*
and *"marray_lookup m xs l = Some (l', ys, i)"*
and *"ys $ i = Some l''"*
shows *"l'' |∈| L"*
⟨*proof*⟩

lemma *range_safe_mlookup:*
assumes *"range_safe s m l = Some L"*
and *"mlookup m xs l = Some l'"*
shows *"l' |∈| L"*
⟨*proof*⟩

lemma *mlookup_range_safe_subs:*
assumes *"mlookup m is l = Some l'"*
and *"range_safe s m l' = Some L"*
and *"range_safe s' m l = Some L'"*
shows *"L |⊆| L'"*
⟨*proof*⟩

lemma *mlookup_range_safe_some:*
assumes *"mlookup m is l = Some l'"*
and *"range_safe s m l = Some L"*
shows *"∃x. m $l' = Some x"*
⟨*proof*⟩

lemma *noloops:*
assumes *"mlookup m (i # is) l = Some l'"*
and *"range_safe s m l = Some L"*

and *"range_safe s m l' = Some L'"*
shows *"l |∉| L'"*
⟨*proof*⟩

definition *range* **where** *"range ≡ range_safe {|l|}"*

lemma *range_subs:*
assumes *"range m l = Some X"*
shows *"l |∈| X"*
⟨*proof*⟩

lemma *range_subs2:*
assumes *"range m l = Some X"*
shows *"fset X ⊆ loc m"*
⟨*proof*⟩

lemma *range_same:*
assumes *"range m l = Some L"*
and *"∀l'|∈|L. m' $ l' = m $ l'"*
shows *"range m' l = Some L"*
⟨*proof*⟩

lemma *range_prefix:*
assumes *"prefix m m'"*
and *"range m l = Some L"*
shows *"range m' l = Some L"*
⟨*proof*⟩

lemma *range_safe_mlookup_range:*
assumes *"range_safe s m l = Some L"*
and *"mlookup m xs l = Some l'"*
shows *"∃L'. range m l' = Some L' ∧ L' |⊆| L"*
⟨*proof*⟩

lemma *range_locations:*
assumes *"range m l = Some L"*
and *"locations m xs l = Some L'"*
shows *"L' |⊆| L"*
⟨*proof*⟩

lemma *range_safe_in_range:*
assumes *"range_safe s m l = Some L"*
and *"l' |∈| L"*
and *"m $ l' = Some (mdata.Array xs)"*
and *"xs $ i = Some i'"*
shows *"∃L'. range m i' = Some L' ∧ L' |⊆| L"*
⟨*proof*⟩

lemma *range_safe_prefix_in_range:*
assumes *"range_safe s m l = Some L"*
and *"l' |∈| L"*
and *"m $ l' = Some (mdata.Array xs)"*
and *"xs $ i = Some i'"*
and *"prefix m m'"*
and *"range m' i' = Some L'"*
shows *"range m i' = Some L'"*
⟨*proof*⟩

lemma *range_mlookup:*
assumes *"range m l = Some L"*
and *"mlookup m xs l = Some l'"*
shows *"l' |∈| L"*
⟨*proof*⟩

lemma *mupdate_range_subset:*
  assumes *"range m l = Some (the (range m l))"*
      and *"m' = m[l':= mdata.Value v]"*
      and *"l' < length m"*
    shows *"∃L. range m' l = Some L ∧ L |⊆| the (range m l)"*
⟨*proof*⟩

3.9 Copy from Memory (Memory)

function *read_safe :: "location fset ⇒ 'v memory ⇒ location ⇒ 'd option"* **where**
  "read_safe s m l =
   (if l |∈| s then None else
      case_memory m l
        (λv. Some (Value v))
        (λxs. those (map (read_safe (finsert l s) m) xs) ⤜ Some ∘ Array))"
⟨*proof*⟩
termination
  ⟨*proof*⟩

lemma *read_safe_cases:*
  assumes *"read_safe s m l = Some c"*
  obtains *(basic) v*
    where *"m $ l = Some (mdata.Value v)"*
      and *"l |∉| s"*
      and *"c = Value v"*
   | (array) xs as
    where *"l |∉| s"*
      and *"m $ l = Some (mdata.Array xs)"*
      and *"those (map (read_safe (finsert l s) m) xs) = Some as"*
      and *"c = Array as"*
  ⟨*proof*⟩

lemma *read_safe_array:*
  assumes *"m0 $ l1 = Some (mdata.Array ls)"*
      and *"read_safe s m0 l1 = Some cd1"*
      and *"ls $ i' = Some l''"*
      and *cd'_def: "read_safe (finsert l1 s) m0 l'' = Some cd'"*
  obtains *cs*
    where *"cd1 = Array cs"*
      and *"cs $ i' = Some cd'"*
      and *"length cs = length ls"*
  ⟨*proof*⟩

lemma *read_safe_update_value:*
  assumes *"read_safe s m l = Some cd"*
      and *"m' = m[l' := mdata.Value v]"*
    shows *"∃cd'. read_safe s m' l = Some cd'"*
⟨*proof*⟩

lemma *read_safe_subset_same:*
  assumes *"read_safe s m l = Some x"*
    and *"s' |⊆| s"*
  shows *"read_safe s' m l = Some x"*
  ⟨*proof*⟩

lemma *read_safe_some_same:*
  assumes *"m $ l1 = m $ l2"*
      and *"read_safe s1 m l1 = Some cd1"*
      and *"read_safe s2 m l2 = Some cd2"*
    shows *"cd1 = cd2"*
  ⟨*proof*⟩

lemma *read_safe_prefix:*
  assumes *"prefix m m'"*

```
      and "read_safe s m l = Some c"
    shows "read_safe s m' l = Some c"
⟨proof⟩

lemma mlookup_read_safe:
  assumes "mlookup m' xs l = Some x"
      and "m' $ x = Some (mdata.Value v)"
      and "read_safe s m' x = Some a"
    shows "a = Value v"
  ⟨proof⟩

lemma mlookup_read_safe_obtain:
  assumes "mlookup m0 (i#is) l1 = Some l1'"
      and "read_safe s m0 l1 = Some cd1"
  obtains ls i' l'' cd'
  where "to_nat i = Some i'"
    and "ls $ i' = Some l''"
    and "mlookup m0 is l'' = Some l1'"
    and "m0 $ l1 = Some (mdata.Array ls)"
    and "read_safe (finsert l1 s) m0 l'' = Some cd'"
⟨proof⟩

definition "read = read_safe {||}"

lemma read_some_same:
  assumes "read_safe s m l = Some x"
  shows "read m l = Some x"
  ⟨proof⟩

lemma read_append:
  assumes "prefix m m'"
      and "read m l = Some c"
    shows "read m' l = Some c"
  ⟨proof⟩
```

3.10 Copy Memory and Memory Locations (Memory)

```
lemma range_safe_read_safe:
  assumes "range_safe s m l = Some L"
    shows "∃x. read_safe s m l = Some x"
  ⟨proof⟩

lemma read_safe_range_safe:
  assumes "read_safe s m l = Some cd"
      and "range_safe s m l = Some L"
      and "∀l'|∈|L. m' $ l' = m $ l'"
    shows "read_safe s m' l = Some cd"
  ⟨proof⟩

lemma read_range:
  assumes "read m l = Some cd"
      and "range m l = Some L"
      and "∀l'|∈|L. m' $ l' = m $ l'"
    shows "read m' l = Some cd"
  ⟨proof⟩

lemma read_safe_range_safe_same:
  assumes "read_safe s m1 l = Some x"
      and "range_safe s m1 l = Some L"
      and "∀l |∈| s' - s. l |∉| L"
  shows "read_safe s' m1 l = Some x"
  ⟨proof⟩

lemma range_read_some:
```

assumes *"read_safe s m0 l0 = Some cd0"*
shows *"∃L. range_safe s m0 l0 = Some L"*
⟨*proof*⟩

lemma *read_safe_range_safe_subs:*
assumes *"m $ l1' = Some (mdata.Array ls)"*
and *"l2 ∈ set ls"*
and *"mlookup m is2 l1 = Some l1'"*
and *"range_safe s m l1 = Some L1"*
and *"read_safe s m l1 = Some cd"*
shows *"∃x y. read_safe s m l2 = Some x ∧ range_safe s m l2 = Some y ∧ y |⊆| L1"*
⟨*proof*⟩

3.11 Separation Check (Memory)

definition *disjoint:: "'v memory ⇒ location fset ⇒ bool"* **where**
"disjoint m L ≡
∀x |∈| L. ∀xs. m$x = Some (mdata.Array xs)
⟶ (∀i j i' j' L L'.
i ≠ j ∧ xs $ i = Some i' ∧ xs$j = Some j' ∧ range m i' = Some L ∧ range m j' = Some L'
⟶ L |∩| L' = {||})"

lemma *disjoint_subs[intro]:*
assumes *"L' |⊆| L"*
and *"disjoint m L"*
shows *"disjoint m L'"*
⟨*proof*⟩

lemma *disjoint_disjoint:*
assumes *"disjoint m L"*
and *"range m l = Some L"*
and *"∀l |∈| L. m $ l = m' $ l"*
shows *"disjoint m' L"*
⟨*proof*⟩

lemma *disjoint_prefix:*
assumes *"fset L ⊆ loc m"*
and *"prefix m m'"*
and *"disjoint m L"*
and *"range_safe s m' l = Some L"*
shows *"disjoint m' L"*
⟨*proof*⟩

lemma *update_some:*
"∀is1 L1 cd1 L3. mlookup m0 is1 l1 = Some l1' ∧
m1 $ l1' = m0 $ l2 ∧
range_safe s m0 l1 = Some L1 ∧
range_safe s m0 l1' = Some L1' ∧
range_safe s2 m0 l2 = Some L2 ∧
(∀l |∈| L1 |-| L1'. m1 $ l = m0 $ l) ∧
(∀l |∈| L2. m1 $ l = m0 $ l) ∧
read_safe s m0 l1 = Some cd1 ∧
read_safe s2 m0 l2 = Some cd2 ∧
s |∩| L2 = {||} ∧
locations m0 is1 l1 = Some L3 ∧
L3 |∩| L2 = {||} ∧
l1' |∉| L2 ∧
disjoint m0 L1
⟶ (∃x. read_safe s m1 l1 = Some x)" (**is** *"?P s m1 l1"*)
⟨*proof*⟩

lemma *update_some_obtains_read:*
assumes *"mlookup m0 is1 l1 = Some l1'"*
and *"m1 $ l1' = m0 $ l2"*

```
  and "range_safe s0 m0 l1 = Some L1"
  and "range_safe s0 m0 l1' = Some L1'"
  and "range_safe s1 m0 l2 = Some L2"
  and "(∀l |∈| L1 |-| L1'. m1 $ l = m0 $ l)"
  and "(∀l |∈| L2. m1 $ l = m0 $ l)"
  and "read_safe s0 m0 l1 = Some cd1"
  and "read_safe s1 m0 l2 = Some cd2"
  and "s0 |∩| L2 = {||}"
  and "locations m0 is1 l1 = Some L3"
  and "L3 |∩| L2 = {||}"
  and "l1' |∉| L2"
  and "disjoint m0 L1"
obtains x where "read_safe s0 m1 l1 = Some x"
```
⟨*proof*⟩

```
lemma update_some_obtains_range:
  assumes "mlookup m0 is1 l1 = Some l1'"
  and "m1 $ l1' = m0 $ l2"
  and "range_safe s0 m0 l1 = Some L1"
  and "range_safe s0 m0 l1' = Some L1'"
  and "range_safe s1 m0 l2 = Some L2"
  and "(∀l |∈| L1 |-| L1'. m1 $ l = m0 $ l)"
  and "(∀l |∈| L2. m1 $ l = m0 $ l)"
  and "s0 |∩| L2 = {||}"
  and "locations m0 is1 l1 = Some L3"
  and "L3 |∩| L2 = {||}"
  and "l1' |∉| L2"
  and "disjoint m0 L1"
obtains L where "range_safe s0 m1 l1 = Some L"
```
⟨*proof*⟩

```
lemma disjoint_range_disj:
  assumes "disjoint m0 L"
      and "x |∈| L"
      and "m0 $ x = Some (mdata.Array xs)"
      and "m0 $ x = m1 $ x"
      and "∀l ∈ set xs. range m1 l = range m0 l"
    shows
  "∀xs. m1$x = Some (mdata.Array xs)
    ⟶ (∀i j i' j' L L'.
         i ≠ j ∧ xs $ i = Some i' ∧ xs$j = Some j' ∧ range m1 i' = Some L ∧ range m1 j' = Some L'
      ⟶ L |∩| L' = {||})"
```
⟨*proof*⟩

```
lemma update_some_range_subset:
  assumes "mlookup m0 is1 l1 = Some l1'"
      and "m1 $ l1' = m0 $ l2'"
      and "range_safe s m0 l1 = Some L1"
      and "range_safe s m0 l1' = Some L1'"
      and "range_safe s m0 l2' = Some L2'"
      and "(∀l |∈| L1 |-| L1'. m1 $ l = m0 $ l)"
      and "(∀l |∈| L2'. m1 $ l = m0 $ l)"
      and "disjoint m0 L1"
      and "range_safe s m1 l1 = Some L"
    shows "L |⊆| L1 |∪| L2'"
```
⟨*proof*⟩

```
lemma disjoint_update:
  assumes "mlookup m0 is1 l1 = Some l1'"
      and "m1 $ l1' = m0 $ l2'"
      and "range_safe s m0 l1 = Some L1"
      and "range_safe s m0 l1' = Some L1'"
      and "range_safe s m0 l2' = Some L2'"
      and "(∀l |∈| L1 |-| L1'. m1 $ l = m0 $ l)"
```

```
      and "(∀ l |∈| L2'. m1 $ l = m0 $ l)"
      and "disjoint m0 L1"
      and "disjoint m0 L2'"
      and "range_safe s m1 l1 = Some L"
      and "L1 |-| L1' |∩| L2' = {||}"
    shows "disjoint m1 L"
```
⟨*proof*⟩

end

3.12 Array Data (Memory)

```
datatype 'v adata =
  is_Value: Value (vt: "'v")
| is_Array: Array (ar: "'v adata list")

abbreviation case_adata where "case_adata cd vf af ≡ adata.case_adata vf af cd"

global_interpretation a_data: data adata.Value adata.Array
  defines aread_safe = a_data.read_safe
      and aread = a_data.read
      and arange_safe = a_data.range_safe
      and arange = a_data.range
      and adisjoint = a_data.disjoint
```
⟨*proof*⟩

3.13 Array Lookup (Memory)

alookup is cd navigates array cd according to the index sequence is.

```
fun alookup :: "'v::vtype list ⇒ 'v adata ⇒ 'v adata option" where
  "alookup [] s = Some s"
| "alookup (i # is) (adata.Array xs) = to_nat i ≫= ($) xs ≫= alookup is"
| "alookup _ _ = None"

lemma alookup_obtains_some:
  assumes "alookup is s = Some sd"
  obtains "is = []" and "sd = s"
  | i is' i' xs sd' where "is = i # is'" and "s = adata.Array xs" and "to_nat i = Some i'" and "xs $
i' = Some sd'" and "alookup is' sd' = Some sd"
```
⟨*proof*⟩

```
lemma alookup_append:
  "alookup (xs1@xs2) cd = alookup xs1 cd ≫= alookup xs2"
```
⟨*proof*⟩

```
lemma alookup_empty_some:
    shows "alookup [] cd = Some cd"
```
⟨*proof*⟩

```
lemma alookup_nempty_some:
  assumes "to_nat x = Some i"
      and "cd = adata.Array a"
      and "i < length a"
      and "alookup xs (a!i) = Some cd'"
    shows "alookup (x # xs) cd = Some cd'"
```
⟨*proof*⟩

```
proposition alookup_same: "(∀ xs. alookup xs cd1 = alookup xs cd2) ≡ cd1 = cd2"
```
⟨*proof*⟩

3.14 Array Lookup and Memory Copy (Memory)

```
lemma read_alookup_obtains:
  assumes "aread_safe s m l = Some cd"
      and "mlookup m xs l = Some l'"
```

shows *"∃cd'. aread_safe s m l' = Some cd' ∧ alookup xs cd = Some cd'"*
⟨*proof*⟩

lemma *mlookup_read_alookup:*
assumes *"mlookup m0 is l1 = Some l1'"*
and *"aread_safe s m0 l1 = Some cd1"*
shows *"∃cd'. alookup is cd1 = Some cd'"*
⟨*proof*⟩

3.15 Array Update (Memory)

fun *aupdate :: "'v::vtype list ⇒ 'v adata ⇒ 'v adata ⇒ 'v adata option"* **where**
"aupdate [] v _ = Some v"
| "aupdate (i # is) v (adata.Array xs)
= to_nat i
≫= (λi. (xs $ i ≫= aupdate is v)
≫= Some ∘ adata.Array ∘ list_update xs i)"
| "aupdate _ _ _ = None"

lemma *aupdate_obtain:*
assumes *"aupdate is v cd = Some cd'"*
obtains
(nil) "is = []" **and** *"cd' = v"*
| (cons) i is' xs i' i'' cd''
where *"is = i # is'"*
and *"cd=adata.Array xs"*
and *"to_nat i = Some i'"*
and *"xs $ i' = Some i''"*
and *"aupdate is' v i'' = Some cd''"*
and *"cd' = adata.Array (list_update xs i' cd'')"*
⟨*proof*⟩

lemma *aupdate_nth_same:*
assumes *"aupdate (i # is) v (adata.Array as) = Some (adata.Array as')"*
and *"to_nat i = Some i'"*
and *"i'' ≠ i'"*
shows *"as ! i'' = as' ! i''"*
⟨*proof*⟩

lemma *aupdate_alookup:*
assumes *"aupdate is v cd = Some cd'"*
shows *"alookup is cd' = Some v"*
⟨*proof*⟩

lemma *alookup_aupdate_alookup:*
assumes *"alookup xs0 cd0 = Some cd0'"*
and *"aupdate xs1 cd0' cd1 = Some cd1'"*
shows *"alookup (xs1@ys) cd1' = alookup (xs0@ys) cd0"*
⟨*proof*⟩

lemma *alookup_update_some:*
assumes *"alookup xs2 cd2 = Some cd"*
and *"alookup xs1 cd1 = Some cd"*
shows *"alookup xs2 cd2 ≫= (λcd. aupdate xs1 cd cd1) = Some cd1"*
⟨*proof*⟩

lemma *alookup_to_nat_same:*
assumes *"map to_nat xs = map to_nat ys"*
shows *"alookup xs cd = alookup ys cd"*
⟨*proof*⟩

lemma *aupdate_alookup_prefix:*
assumes *"ys = xs' @ zs"*
and *"map to_nat xs = map to_nat xs'"*

```
      and "aupdate xs v cd = Some cd'"
    shows "alookup ys cd' = alookup zs v"
```
⟨*proof*⟩

lemma *aupdate_alookup_nprefix1:*
```
  assumes "xs = ys @ zs"
      and "aupdate xs v cd = Some cd'"
    shows "alookup ys cd' = alookup ys cd ≫= aupdate zs v"
```
⟨*proof*⟩

lemma *aupdate_alookup_nprefix2:*
```
  assumes "xs = ys' @ zs"
      and "map to_nat ys = map to_nat ys'"
      and "aupdate xs v cd = Some cd'"
    shows "alookup ys cd' = alookup ys cd ≫= aupdate zs v"
```
⟨*proof*⟩

lemma *updateCalldata_clookup_nprefix:*
```
  assumes "aupdate (x#xs) v cd = Some cd'"
      and "to_nat x ≠ to_nat y"
    shows "alookup (y#zs) cd' = alookup (y#zs) cd"
```
⟨*proof*⟩

lemma *aupdate_alookup_nprefix3:*
```
  assumes "∄xs'. map to_nat ys = map to_nat xs @ xs'"
      and "∄ys'. map to_nat xs = map to_nat ys @ ys'"
      and "aupdate xs v cd = Some cd'"
    shows "alookup ys cd' = alookup ys cd"
```
⟨*proof*⟩

lemma *alookup_aupdate_some:*
```
  assumes "∃x. alookup xs cd = Some x"
    shows "∃x. aupdate xs v cd = Some x"
```
⟨*proof*⟩

3.16 Calldata Update and Memory Copy (Memory)

lemma *separate_memory:*
```
  assumes "mlookup m xs1 l1 = Some l1'"
  and  "mlookup m xs2 l2 = Some l2'"
  and "m $ l1' = m $ l2'"
  and "aread_safe s1 m l1 = Some cd1"
  and "aread_safe s2 m l2 = Some cd2"
shows "alookup xs2 cd2 ≫= (λcd. aupdate xs1 cd cd1) = Some cd1"
```
⟨*proof*⟩

lemma *split_memory:*
```
  assumes "aread_safe s1 m l = Some cd"
      and "mlookup m xs l = Some l'"
    shows "aread_safe s1 m l' ≫= (λcd'. aupdate xs cd' cd) = Some cd"
```
⟨*proof*⟩

lemma *mlookup_read_update:*
```
  assumes "mlookup m0 is l1 = Some l1'"
      and "aread_safe s m0 l1 = Some cd1"
    shows "∃cd'.
           aupdate is cd cd1 = Some cd' ∧
             (is ≠ [] ⟶
               (∃ls. m0 $ l1 = Some (mdata.Array ls)
               ∧ (∃as. cd' = adata.Array as ∧ length as = length ls)))"
```
⟨*proof*⟩

lemma *read_safe_lookup_update_value:*
```
  assumes "mlookup m0 is1 l1 = Some l1'"
```

```
      and "l1' < length m0"
      and "m1 = m0[l1':=mdata.Value v]"
      and "arange_safe s m0 l1 = Some L1"
      and "arange_safe s m0 l1' = Some L1'"
      and "∀l |∈| L1 |-| L1'. m1 $ l = m0 $ l"
      and "aread_safe s m0 l1 = Some cd0"
      and "adisjoint m0 L1"
      and "aread_safe s m1 l1 = Some cd1"
    shows "aupdate is1 (Value v) cd0 = Some cd1"
```
⟨*proof*⟩

lemma *read_safe_lookup_update:*
```
  assumes "mlookup m0 is1 l1 = Some l1'"
      and "mlookup m0 is2 l2 = Some l2'"
      and "m1 $ l1' = m0 $ l2'"
      and "arange_safe s m0 l1 = Some L1"
      and "arange_safe s m0 l1' = Some L1'"
      and "arange_safe s2 m0 l2 = Some L2"
      and "(∀l |∈| L1 |-| L1'. m1 $ l = m0 $ l)"
      and "(∀l |∈| L2. m1 $ l = m0 $ l)"
      and "aread_safe s m0 l1 = Some cd1"
      and "aread_safe s2 m0 l2 = Some cd2"
      and "adisjoint m0 L1"
      and "aread_safe s m1 l1 = Some cd"
    shows "alookup is2 cd2 ⤜ (λcd. aupdate is1 cd cd1) = Some cd"
```
⟨*proof*⟩

lemma *range_safe_update_some:*
```
  assumes "mlookup m0 is1 l1 = Some l1'"
      and "m1 $ l1' = m0 $ l2'"
      and "arange_safe s m0 l1 = Some L1"
      and "arange_safe s m0 l1' = Some L1'"
      and "arange_safe s2 m0 l2' = Some L2'"
      and "(∀l |∈| L1 |-| L1'. m1 $ l = m0 $ l)"
      and "(∀l |∈| L2'. m1 $ l = m0 $ l)"
      and "adisjoint m0 L1"
      and "s |∩| L2' = {||}"
      and "the (locations m0 is1 l1) |∩| L2' = {||}"
      and "l1' |∉| L2'"
    shows "∃L. arange_safe s m1 l1 = Some L"
```
⟨*proof*⟩

lemma *range_update_some:*
```
  assumes "mlookup m0 is1 l1 = Some l1'"
      and "m1 $ l1' = m0 $ l2'"
      and "arange m0 l1 = Some L1"
      and "arange m0 l1' = Some L1'"
      and "arange m0 l2' = Some L2'"
      and "(∀l |∈| L1 |-| L1'. m1 $ l = m0 $ l)"
      and "(∀l |∈| L2'. m1 $ l = m0 $ l)"
      and "adisjoint m0 L1"
      and "the (locations m0 is1 l1) |∩| L2' = {||}"
      and "l1' |∉| L2'"
    shows "∃L. arange m1 l1 = Some L"
```
⟨*proof*⟩

lemma *range_safe_update_subs:*
```
  assumes "mlookup m0 is1 l1 = Some l1'"
      and "m1 $ l1' = m0 $ l2'"
      and "arange_safe s m0 l1 = Some L1"
      and "arange_safe s m0 l1' = Some L1'"
      and "arange_safe s2 m0 l2' = Some L2'"
      and "(∀l |∈| L1 |-| L1'. m1 $ l = m0 $ l)"
      and "(∀l |∈| L2'. m1 $ l = m0 $ l)"
```

```
      and "adisjoint m0 L1"
      and "arange_safe s m1 l1 = Some L"
    shows "L |⊆| L1 |∪| L2'"
```
⟨*proof*⟩

lemma *range_update_subs:*
```
  assumes "mlookup m0 is1 l1 = Some l1'"
      and "m1 $ l1' = m0 $ l2'"
      and "arange m0 l1 = Some L1"
      and "arange m0 l1' = Some L1'"
      and "arange m0 l2' = Some L2'"
      and "(∀l |∈| L1 |-| L1'. m1 $ l = m0 $ l)"
      and "(∀l |∈| L2'. m1 $ l = m0 $ l)"
      and "adisjoint m0 L1"
      and "arange m1 l1 = Some L"
    shows "L |⊆| L1 |∪| L2'"
```
⟨*proof*⟩

3.17 Initialize Memory (Memory)

function *"write" :: "'v adata ⇒ 'v memory ⇒ location × 'v memory"* **where**
```
  "write (adata.Value x) m = length_append m (mdata.Value x)"
| "write (adata.Array ds) m = (let (ns, m') = fold_map write ds m in (length_append m' (mdata.Array
ns)))"
```
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *write_sprefix: "sprefix m0 (snd (write cd m0))"*
⟨*proof*⟩

lemma *loc_write_take[simp]:*
```
  assumes "i ≤ j"
      and "j < length ds"
    shows "loc (snd (fold_map write (take i ds) m0)) ⊆ loc (snd (fold_map write (take j ds) m0))"
```
⟨*proof*⟩

lemma *write_fold_map_sprefix:*
```
  assumes "ds ≠ []"
  shows "sprefix m0 (snd (fold_map write ds m0))"
```
⟨*proof*⟩

lemma *write_fold_map_mono:*
```
  assumes "prefix ds' ds"
  shows "prefix (snd (fold_map write ds' m0)) (snd (fold_map write ds m0))"
```
⟨*proof*⟩

lemma *write_fold_map_smono:*
```
  assumes "sprefix ds' ds"
  shows "sprefix (snd (fold_map write ds' m0)) (snd (fold_map write ds m0))"
```
⟨*proof*⟩

lemma *write_prefix_mono:*
```
  assumes "prefix ds' ds"
  shows
    "prefix
      (butlast (snd (write (adata.Array ds') m0)))
      (butlast (snd (write (adata.Array ds) m0)))"
```
⟨*proof*⟩

lemma *write_prefix_smono:*
```
  assumes "sprefix ds' ds"
  shows
    "sprefix
```

```
      (butlast (snd (write (adata.Array ds') m0)))
      (butlast (snd (write (adata.Array ds) m0)))"
  ⟨proof⟩

lemma write_length_inc: "length (snd (write cd m0)) > length m0"
  ⟨proof⟩

lemma write_Array_take_Suc:
  assumes "n < length ds"
    shows "fst (write (adata.Array (take (Suc n) ds)) m0)
          = length (snd (write (ds ! n) (butlast (snd (write (adata.Array (take n ds)) m0)))))"
  ⟨proof⟩

lemma butlast_write[simp]:
  "butlast (snd (write (adata.Array ds) m0)) = snd (fold_map write ds m0)"
  ⟨proof⟩

lemma write_sprefix_take:
  assumes "n < length ds"
  shows
    "sprefix
      (snd (write (ds!n) (snd (fold_map write (take n ds) m0))))
      (snd (write (Array ds) m0))"
⟨proof⟩

lemma write_length_suc: "length (snd (write ds m)) = Suc (fst (write ds m))"
  ⟨proof⟩

lemma write_length_suc2:
  assumes "write ds m0 = (l, m)"
  shows "l = length m - 1"
  ⟨proof⟩

lemma write_fold_map_less:
  assumes "n < length (fst (fold_map write ds m))"
  shows "fst (fold_map write ds m) ! n < fst (write (Array ds) m)"
  ⟨proof⟩

lemma write_obtain:
  obtains xs
    where "snd (write (Array ds) m0) $ fst (write (Array ds) m0) = Some (mdata.Array xs)"
      and "length xs = length ds"
      and "∀n < length xs. xs!n < fst (write (Array ds) m0)
          ∧ xs!n = fst (write (ds!n) (snd (fold_map write (take n ds) m0)))
          ∧ prefix (snd (write (ds!n) (snd (fold_map write (take n ds) m0)))) (snd (write (Array ds)
m0))"
⟨proof⟩

lemma write_array_less:
  assumes "write cd m = (l, m')"
      and "m'$l = Some (mdata.Array xs)"
      and "xs $ i = Some l'"
    shows "l' < l"
  ⟨proof⟩

lemma range_notin_s:
  assumes "n < length ds"
      and "n < length xs"
      and "∀n < length xs.
            xs!n < fst (write (Array ds) m0) ∧
            xs!n = fst (write (ds!n) (snd (fold_map write (take n ds) m0)))"
      and "∀l≥length m0. l < length (snd (write (adata.Array ds) m0)) ⟶ l |∉| s"
    shows "∀l≥length (snd (fold_map write (take n ds) m0)).
            l < length (snd (write (ds!n) (snd (fold_map write (take n ds) m0))))
```

```
          ⟶ l |∉| (finsert (fst (write (adata.Array ds) m0)) s)"
```
⟨*proof*⟩

lemma *length_write_write:*
```
  assumes "n < length ds"
      and "∀l≥length m0. l < length (snd (write (adata.Array ds) m0)) ⟶ l |∉| s"
      and "xs!n < fst (write (Array ds) m0)"
      and "xs!n = fst (write (ds!n) (snd (fold_map write (take n ds) m0)))"
  shows "∀l≥length (butlast (snd (write (adata.Array (take n ds)) m0))).
         l < length (snd (write (ds ! n) (butlast (snd (write (adata.Array (take n ds)) m0))))) ⟶
         l |∉| finsert (fst (write (adata.Array ds) m0)) s"
```
⟨*proof*⟩

lemma *marray_lookup_write_take:*
```
  assumes "is ≠ []"
      and "write (adata.Array ds) m = (l, m')"
      and "m' $ l = Some (mdata.Array ns)"
      and "ns $ i = Some l'"
      and "marray_lookup m'' is l' = Some (lx, nsx, ix)"
      and "prefix m' m''"
    shows "marray_lookup (snd (fold_map write (take (Suc i) ds) m)) is l' = marray_lookup m'' is l'"
```
⟨*proof*⟩

lemma *locations_lookup_write_take:*
```
  assumes "write (adata.Array ds) m = (l, m')"
      and "m' $ l = Some (mdata.Array ns)"
      and "ns $ i = Some l'"
      and "locations m'' is l' = Some L"
      and "prefix m' m''"
    shows "locations (snd (fold_map write (take (Suc i) ds) m)) is l' = locations m'' is l'"
```
⟨*proof*⟩

3.18 Memory Init and Lookup (Memory)

lemma *marray_lookup_write_less:*
```
  assumes "is ≠ []"
    and "write cd m = (l, m')"
    and "marray_lookup m' is l = Some (lx, nsx, ix)"
    and "nsx $ ix = Some l'"
  shows "l' < l"
```
⟨*proof*⟩

lemma *write_marray_lookup_locations:*
```
  assumes "write cd m = (l, m')"
      and "marray_lookup m' xs l = Some (l1, xs1, i1)"
      and "xs1 $ i1 = Some l2"
      and "locations m' xs l = Some L"
    shows "l2 |∉| L"
```
⟨*proof*⟩

lemma *write_lookup_some:*
```
  assumes "xs ≠ []"
      and "write cd m = (l, m')"
      and "alookup xs cd = Some x"
      and "prefix m' m''"
    shows "∃lz xsz iz z. marray_lookup m'' xs l = Some (lz, xsz, iz) ∧ xsz $ iz = Some z"
```
⟨*proof*⟩

lemma *mlookup_some:*
```
  assumes "write cd m = (l, m')"
      and "alookup xs cd = Some x"
    shows "∃y. mlookup m' xs l = Some y"
```
⟨*proof*⟩

lemma *write_mlookup_locations:*
assumes *"write cd m = (l, m')"*
and *"mlookup m' xs l = Some l1"*
and *"locations m' xs l = Some L"*
shows *"l1 |∉| L"*
⟨*proof*⟩

lemma *write_locations_some:*
assumes *"write cd m = (l, m')"*
and *"alookup xs cd = Some x"*
and *"prefix m' m''"*
shows *"∃y. locations m'' xs l = Some y"*
⟨*proof*⟩

3.19 Memory Init and Memory Locations (Memory)

lemma *write_range_safe_in:*
assumes *"write (adata.Array ds) m0 = (l, m)"*
and *"arange_safe s m l = Some L"*
and *"x |∈| L"*
shows *"x = l ∨*
(∃n y L'. n<length ds ∧ fst (write (ds ! n) (snd (fold_map write (take n ds) m0)))
= y ∧ arange_safe s m y = Some L' ∧ x |∈| L')"
⟨*proof*⟩

theorem *write_arange_safe:*
assumes *"∀l ≥ length m0. l < length (snd (write cd m0)) ⟶ ¬ l |∈| s"*
shows *"s_disj_fs (loc m0) (arange_safe s (snd (write cd m0)) (fst (write cd m0)))"*
⟨*proof*⟩

corollary *write_arange:*
assumes *"write cd m0 = (l, m)"*
shows *"s_disj_fs (loc m0) (arange m l)"*
⟨*proof*⟩

lemma *fold_map_write_arange:*
assumes *"write (adata.Array ds) m0 = (l, m)"*
and *"j < length ds"*
and *"i < j"*
shows *"s_disj_fs*
(loc (snd (write (ds ! i) (snd (fold_map write (take i ds) m0)))))
(arange m (fst (write (ds ! j) (snd (fold_map write (take j ds) m0)))))"
⟨*proof*⟩

theorem *write_loc_safe:*
assumes *"∀l ≥ length m0. l < length (snd (write cd m0)) ⟶ ¬ l |∈| s"*
shows
"s_union_fs
(loc (snd (write cd m0)))
(loc m0)
(arange_safe s (snd (write cd m0)) (fst (write cd m0)))
∧ (∀x |∈| the (arange_safe s (snd (write cd m0)) (fst (write cd m0))).
x < (length (snd (write cd m0))))"
⟨*proof*⟩

corollary *write_loc:*
assumes *"write cd m0 = (l, m)"*
shows *"s_union_fs (loc m) (loc m0) (arange m l)"*
⟨*proof*⟩

lemma *fold_map_write_loc:*
assumes *"write (adata.Array ds) m0 = (l, m)"*
and *"i < length ds"*
and *"i' = fst (write (ds ! i) (snd (fold_map write (take i ds) m0)))"*

```
   shows "fs_subs_s
            (arange m i')
            (loc (snd (write (ds ! i ) (snd (fold_map write (take i ds) m0)))))"
```
⟨proof⟩

```
lemma prefix_write_range_safe_same:
  assumes "prefix (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) m"
      and "arange_safe s m y = Some L'"
      and "y = fst (write (ds ! n) (snd (fold_map write (take n ds) m0)))"
      and "∀l≥length (snd (fold_map write (take n ds) m0)).
            l < length (snd (write (ds ! n) (snd (fold_map write (take n ds) m0))))
          ⟶ l |∉| s"
    shows "arange_safe s (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) y = Some L'"
```
⟨proof⟩

```
lemma prefix_write_nth_same:
  assumes "m $ x = Some (mdata.Array xs)"
      and "fst (write (ds ! n) (snd (fold_map write (take n ds) m0))) = y"
      and "arange_safe s m y = Some L'"
      and "x |∈| L'"
      and "prefix (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) m"
    shows "snd (write (ds ! n) (snd (fold_map write (take n ds) m0))) $ x = Some (mdata.Array xs)"
```
⟨proof⟩

3.20 Memory Init and Memory Copy (Memory)

```
theorem write_aread_safe:
  assumes "∀l ≥ length m0. l < length (snd (write cd m0)) ⟶ ¬ l |∈| s"
  shows "∀mx. prefix (snd (write cd m0)) mx
        ⟶ aread_safe s mx (fst (write cd m0)) = Some cd"
```
⟨proof⟩

```
corollary write_read:
  assumes "write cd m0 = (l, m)"
      and "prefix m mx"
    shows "aread mx l = Some cd"
```
⟨proof⟩

3.21 Minit and Separation Check (Memory)

```
lemma write_adisjoint_safe:
  assumes "write cd m0 = (l, m1)"
      and "arange_safe s m1 l = Some L"
      and "∀l ≥ length m0. l < length (snd (write cd m0)) ⟶ ¬ l |∈| s"
    shows "adisjoint m1 L"
```
⟨proof⟩

```
corollary write_adisjoint:
  assumes "write a m = (l, m')"
      and "arange m' l = Some L"
    shows "adisjoint m' L"
```
⟨proof⟩

```
end
theory Stores
  imports Memory
begin
```

3.22 Calldata (Stores)

```
datatype 'v call_data =
  is_Value: Value (vt: "'v")
  | is_Array: Array (ar: "'v call_data list")
```

fun *c_to_s* **where**
  "c_to_s (Value v) = adata.Value v"
| *"c_to_s (Array xs) = adata.Array (map c_to_s xs)"*

fun *s_to_c* **where**
  "s_to_c (adata.Value v) = Value v"
| *"s_to_c (adata.Array xs) = Array (map s_to_c xs)"*

lemma *stoc_ctos:*
  "s_to_c (c_to_s a) = a"
  ⟨*proof*⟩

lemma *eq_iff_stoc:*
  "a = b ⟷ s_to_c a = s_to_c b"
⟨*proof*⟩

function *(sequential) T* **where**
  "T (adata.Value v) (Value v') = (v = v')"
 |*"T (adata.Array xs) (Array xs') = list_all2 T xs xs'"*
 |*"T _ _ = False"*
  ⟨*proof*⟩
termination *T*
  ⟨*proof*⟩

lemma *T_eq_stoc:*
  "T x y = (s_to_c x = y)"
⟨*proof*⟩

lemma *q: "Quotient (=) s_to_c c_to_s T"*
⟨*proof*⟩

setup_lifting *q*

code_datatype *call_data.Value call_data.Array*

lift_definition *"write" :: "'v call_data ⇒ 'v memory ⇒ nat × 'v memory"* **is** *Memory.write* ⟨*proof*⟩
lift_definition *clookup :: "'v::vtype list ⇒ 'v call_data ⇒ 'v call_data option"* **is** *Memory.alookup*
⟨*proof*⟩

context includes *lifting_syntax* **begin**

lemma *Value_transfer[transfer_rule]: "(R ===> (pcr_call_data R)) adata.Value call_data.Value"*
  ⟨*proof*⟩

lemma *Array_transfer[transfer_rule]: "((list_all2 (pcr_call_data R)) ===> (pcr_call_data R))*
adata.Array call_data.Array"
  ⟨*proof*⟩

lemma *fold_map_transfer[transfer_rule]: "((A ===> B ===> rel_prod C B) ===> list_all2 A ===> B ===>*
rel_prod (list_all2 C) B) fold_map fold_map"
  ⟨*proof*⟩

lemma *eq_transfer[transfer_rule]: "(rel_option (pcr_call_data (=)) ===> (rel_option (pcr_call_data*
(=)) ===> (=))) (=) (=)"
  ⟨*proof*⟩

lemma *nth_safe_transfer[transfer_rule]: "(list_all2 ((pcr_call_data (=))) ===> ((=)) ===> rel_option*
(pcr_call_data (=))) nth_safe nth_safe"
  ⟨*proof*⟩

end

lemma *write_length_append[simp,code]: "write (call_data.Value x) m = length_append m (mdata.Value x)"*
  ⟨*proof*⟩

lemma *write_fold_map_length_append[simp,code]:*
```
"write (call_data.Array ds) m = (let (ns, m')
  = fold_map write ds m in (length_append m' (mdata.Array ns)))"
```
⟨*proof*⟩

lemma *clookup[simp,code]:*
```
"clookup [] s = Some s"
```
⟨*proof*⟩

lemma *clookup2[simp,code]:*
```
"clookup (i # is) (call_data.Array xs) = vtype_class.to_nat i ≫= ($) xs ≫= clookup is"
```
⟨*proof*⟩

lemma *clookup_none[simp,code]:*
```
"clookup (v # va) (call_data.Value vb) = None"
```
⟨*proof*⟩

lemma *write_length_inc: "length (snd (write cd m0)) > length m0"*
⟨*proof*⟩

corollary *write_loc:*
assumes *"write cd m0 = (l, m)"*
shows *"s_union_fs (loc m) (loc m0) (arange m l)"*
⟨*proof*⟩

3.23 Storage (Stores)

datatype *'v storage_data =*
```
  is_Value: Value (vt:'v)
| is_Array: Array (ar: "'v storage_data list")
| is_Map: Map (mp: "'v ⇒ 'v storage_data")
```

abbreviation *storage_disjoint* **where** *"storage_disjoint sd vf af mf ≡ case_storage_data vf af mf sd"*

3.24 Storage Lookup (Stores)

slookup is sd navigates storage sd according to the index sequence is.

fun *slookup :: "'v::vtype list ⇒ 'v storage_data ⇒ 'v storage_data option"* **where**
```
  "slookup [] s = Some s"
| "slookup (i # is) (storage_data.Array xs) = to_nat i ≫= ($) xs ≫= slookup is"
| "slookup (i # is) (storage_data.Map f) = slookup is (f i)"
| "slookup _ _ = None"
```

3.25 Storage Update (Stores)

fun *updateStore :: "('v::vtype) list ⇒ ('v storage_data ⇒ 'v storage_data) ⇒ 'v storage_data ⇒ 'v storage_data option"* **where**
```
  "updateStore [] f v = Some (f v)"
| "updateStore (i # is) f (storage_data.Array xs) =
    to_nat i
    ≫= (λi. xs $ i ≫= updateStore is f ≫= list_update_safe xs i ≫= Some ∘ storage_data.Array)"
| "updateStore (i # is) f (Map m) = updateStore is f (m i) ≫= Some ∘ storage_data.Map ∘ fun_upd m i"
| "updateStore _ _ _ = None"
```

3.25.1 Copy from Calldata to Memory

fun *read_calldata_memory :: "'v call_data ⇒ location ⇒ 'v memory ⇒ (location × 'v mdata × 'v memory)"* **where**
```
  "read_calldata_memory (call_data.Value x) p m = (p, mdata.Value x, m)"
| "read_calldata_memory (call_data.Array ds) p m =
      (let (ns, m') = fold_map write ds m in (p, mdata.Array ns, m'))"
```

3.25.2 Copy from Calldata to Storage

fun *read_calldata_storage :: "'v call_data ⇒ 'v storage_data"* **where**

```
  "read_calldata_storage (call_data.Value v) = storage_data.Value v"
| "read_calldata_storage (call_data.Array xs) = storage_data.Array (map read_calldata_storage xs)"
```

3.25.3 Copy from Storage to Memory

fun *convert2 :: "'v storage_data ⇒ 'v adata option"* **where**
```
  "convert2 (storage_data.Value x) = Some (adata.Value x)"
| "convert2 (storage_data.Array ds) = those (map convert2 ds) ≫= Some ∘ adata.Array"
| "convert2 _ = None"
```

fun *convert :: "'v storage_data ⇒ 'v call_data option"* **where**
```
  "convert (storage_data.Value x) = Some (call_data.Value x)"
| "convert (storage_data.Array ds) = those (map convert ds) ≫= Some ∘ call_data.Array"
| "convert _ = None"
```

definition *read_storage_memory :: "'v storage_data ⇒ location ⇒ 'v memory ⇒ (location × 'v mdata × 'v memory) option"* **where**
```
  "read_storage_memory sd p m =
    do {
      cd ← convert sd;
      Some (read_calldata_memory cd p m)
    }"
```

global_interpretation *storage_data: data storage_data.Value storage_data.Array*
  defines *read_storage_safe = storage_data.read_safe*
    and *read_storage = storage_data.read*
    and *range_storage_safe = storage_data.range_safe*
    and *range_storage = storage_data.range*
  ⟨*proof*⟩

3.25.4 Data type

record *'v pointer =*
```
  Location :: String.literal
  Offset :: "'v list"
```

datatype *'v kdata =*
```
  Storage "'v pointer option" |
  Memory (memloc: location) |
  Calldata "'v pointer option" |
  Value (vt: "'v")
```

end
theory *State*
imports *Stores "HOL-Library.Word"*
begin

3.26 Value types (State)

type_synonym *bytes = string*
type_synonym *id = String.literal*

datatype *('a::address) valtype =*
```
  Bool (bool: bool)
| Uint (uint: "256 word")
| Address (ad: 'a)
| Bytes bytes
```
— bytes1, ..., bytes32

instantiation *valtype :: (address) vtype*
begin

fun *to_nat_valtype::"'a valtype ⇒ nat option"* **where**
```
  "to_nat_valtype (Uint x) = Some (unat x)"
```

```
| "to_nat_valtype _ = None"
```

instance ⟨*proof*⟩

end

3.27 Common functions (State)

```
fun lift_bool_unary::"(bool ⇒ bool) ⇒ ('a::address) valtype ⇒ ('a::address) valtype option" where
  "lift_bool_unary op (Bool b) = Some (Bool (op b))"
| "lift_bool_unary _ _ = None"

definition vtnot where
  "vtnot = lift_bool_unary Not"

fun lift_bool_binary::"(bool ⇒ bool ⇒ bool) ⇒ ('a::address) valtype ⇒ ('a::address) valtype ⇒
('a::address) valtype option" where
  "lift_bool_binary op (Bool l) (Bool r) = Some (Bool (op l r))"
| "lift_bool_binary _ _ _ = None"

definition vtand where
  "vtand = lift_bool_binary (∧)"

definition vtor where
  "vtor = lift_bool_binary (∨)"

fun vtequals where
  "vtequals (Uint l) (Uint r) = Some (Bool (l = r))"
| "vtequals (Address l) (Address r) = Some (Bool (l = r))"
| "vtequals (Bool l) (Bool r) = Some (Bool (l = r))"
| "vtequals (Bytes l) (Bytes r) = Some (Bool (l = r))"
| "vtequals _ _ = None"

fun lift_int_comp::"(256 word ⇒ 256 word ⇒ bool) ⇒ ('a::address) valtype ⇒ ('a::address) valtype ⇒
('a::address) valtype option" where
  "lift_int_comp op (Uint l) (Uint r) = Some (Bool (op l r))"
| "lift_int_comp _ _ _ = None"

definition vtless where
  "vtless = lift_int_comp (<)"

fun lift_int_binary::"(256 word ⇒ 256 word ⇒ 256 word) ⇒ ('a::address) valtype ⇒ ('a::address)
valtype ⇒ ('a::address) valtype option" where
  "lift_int_binary op (Uint l) (Uint r) = Some (Uint (op l r))"
| "lift_int_binary _ _ _ = None"

definition vtplus where
  "vtplus = lift_int_binary (+)"

fun vtplus_safe::"('a::address) valtype ⇒ ('a::address) valtype ⇒ ('a::address) valtype option"
where
  "vtplus_safe (Uint l) (Uint r) = (if unat l + unat r < 2^256 then Some (Uint (l + r)) else None)"
| "vtplus_safe _ _ = None"

declare vtplus_safe.simps[simp del]

definition vtminus where
  "vtminus = lift_int_binary (-)"

fun vtminus_safe::"('a::address) valtype ⇒ ('a::address) valtype ⇒ ('a::address) valtype option"
where
  "vtminus_safe (Uint l) (Uint r) = (if r ≤ l then Some (Uint (l - r)) else None)"
| "vtminus_safe _ _ = None"
```

```
declare vtminus_safe.simps[simp del]

definition vtmult where
  "vtmult = lift_int_binary (*)"

fun vtmult_safe::"('a::address) valtype ⇒ ('a::address) valtype ⇒ ('a::address) valtype option"
where
  "vtmult_safe (Uint l) (Uint r) = (if unat l * unat r < 2^256 then Some (Uint (l * r)) else None)"
| "vtmult_safe _ _ = None"

declare vtmult_safe.simps[simp del]

definition vtmod where
  "vtmod = lift_int_binary (mod)"
```

3.28 Operations on bytes (State)

```
fun vtbytes_index :: "('a::address) valtype ⇒ ('a::address) valtype ⇒ ('a::address) valtype option"
where
  "vtbytes_index (Bytes xs) (Uint i) = (if unat i < length xs then Some (Bytes [xs ! unat i]) else
None)"
| "vtbytes_index _ _ = None"

definition zipWith :: "('a ⇒ 'b ⇒ 'c) ⇒ 'a list ⇒ 'b list ⇒ 'c list" where
  "zipWith op xs ys = map (λ (x, y). op x y) (zip xs ys)"

fun lift_bytes_binary::"(char ⇒ char ⇒ char) ⇒ ('a::address) valtype ⇒ ('a::address) valtype ⇒
('a::address) valtype option" where
  "lift_bytes_binary op (Bytes l) (Bytes r) = (if length l = length r then Some (Bytes (zipWith op l r))
else None)"
| "lift_bytes_binary _ _ _ = None"

fun lift_bytes_unary::"(char ⇒ char) ⇒ ('a::address) valtype ⇒ ('a::address) valtype option" where
  "lift_bytes_unary op (Bytes l) = Some (Bytes (map op l))"
| "lift_bytes_unary _ _ = None"

definition word8_to_char :: "8 word ⇒ char" where
  "word8_to_char w = char_of (unat w)"

definition char_to_word8 :: "char ⇒ 8 word" where
  "char_to_word8 c = of_nat (of_char c)"

definition op_word8_to_char :: "(8 word ⇒ 8 word ⇒ 8 word) ⇒ (char ⇒ char ⇒ char)" where
  "op_word8_to_char op x y = word8_to_char (op (char_to_word8 x) (char_to_word8 y))"

context
  includes bit_operations_syntax
begin

definition vtbytes_and where
  "vtbytes_and = lift_bytes_binary (op_word8_to_char (AND))"

definition vtbytes_or where
  "vtbytes_or = lift_bytes_binary (op_word8_to_char (OR))"

definition vtbytes_xor where
  "vtbytes_xor = lift_bytes_binary (op_word8_to_char (XOR))"

definition vtbytes_not where
  "vtbytes_not = lift_bytes_unary (λ x. word8_to_char ((NOT) (char_to_word8 x)))"

end

fun resize_list :: "nat ⇒ 'a ⇒ 'a list ⇒ 'a list" where
```

```
  "resize_list m pad xs =
     (if length xs < m
      then xs @ replicate (m - length xs) pad
      else take m xs)"

fun vtbytes_cast :: "nat ⇒ ('a::address) valtype ⇒ ('a::address) valtype option" where
  "vtbytes_cast m (Bytes xs) = Some (Bytes (resize_list m (CHR 0x00) xs))"
| "vtbytes_cast m _ = None"
```

3.29 State (State)

3.29.1 Definition

```
type_synonym 'v stack = "(id, 'v kdata) fmap"
type_synonym 'a balances = "'a ⇒ nat"
type_synonym ('a, 'v) storage = "'a ⇒ id ⇒ 'v storage_data"
type_synonym 'v calldata = "(id, 'v call_data) fmap"

record ('a::address) state =
  Memory:: "('a::address valtype) memory"
  Calldata:: "('a::address valtype) calldata"
  Storage:: "('a::address, 'a::address valtype) storage"
  Stack:: "('a::address valtype) stack"
  Balances::"('a::address) balances"

definition sameState where "sameState s s' ≡ state.Stack s' = state.Stack s ∧ state.Memory s' =
state.Memory s ∧ state.Calldata s' = state.Calldata s"
```

3.29.2 Update Function

```
datatype ex = Err

definition balances_update:: "('a::address) ⇒ nat ⇒ ('a::address) state ⇒ ('a::address) state" where
  "balances_update i n s = s(|Balances := (Balances s)(i := n)|)"

definition calldata_update:: "id ⇒ ('a::address valtype) call_data ⇒ ('a::address) state ⇒
('a::address) state" where
  "calldata_update i d = Calldata_update (fmupd i d)"

definition stack_update:: "id ⇒ ('a::address valtype) kdata ⇒ ('a::address) state ⇒ ('a::address)
state" where
  "stack_update i d = Stack_update (fmupd i d)"

definition memory_update:: "location ⇒ ('a::address valtype) mdata ⇒ ('a::address) state ⇒
('a::address) state" where
  "memory_update i d s = s(|Memory := (Memory s)[i := d]|)"

lemma balances_update_id[simp]: "balances_update x (Balances s x) s = s"
  ⟨proof⟩

end
```

4 Expressions and Statements

In this chapter, we formalize expressions and statements as shallow embeddings for Isabelle/HOL. semantics.
theory *Solidity*
imports *State_Monad State "Finite-Map-Extras.Finite_Map_Extras"*
begin

4.1 Value types (Solidity)

```
datatype ('a) rvalue =
  Storage "'a valtype pointer option" |
  Memory (memloc: location) |
  Calldata "'a valtype pointer option" |
  Value (vt: "'a valtype") |
  Empty

definition kdbool where
  "kdbool = Value ∘ Bool"

definition kdSint where
  "kdSint ≡ Value ∘ Uint"

definition kdAddress where
  "kdAddress = Value ∘ Address"

definition kdBytes where
  "kdBytes ≡ Value ∘ Bytes"

fun lift_value_unary::"('a::address) valtype ⇒ ('a::address) valtype option) ⇒ ('a::address) rvalue
⇒ ('a::address) rvalue option" where
  "lift_value_unary op (rvalue.Value v) = op v ⤜ Some ∘ rvalue.Value"
| "lift_value_unary op _ = None"

definition kdnot::"('a::address) rvalue ⇒ ('a::address) rvalue option" where
  "kdnot = lift_value_unary vtnot"

fun lift_value_binary::"(('a::address) valtype ⇒ ('a::address) valtype ⇒ ('a::address) valtype
option) ⇒ ('a::address) rvalue ⇒ ('a::address) rvalue ⇒ ('a::address) rvalue option" where
  "lift_value_binary op (rvalue.Value l) (rvalue.Value r) = op l r ⤜ Some ∘ rvalue.Value"
| "lift_value_binary op _ _ = None"

definition kdequals where
  "kdequals = lift_value_binary vtequals"

definition kdless where
  "kdless = lift_value_binary vtless"

definition kdand where
  "kdand = lift_value_binary vtand"

definition kdor where
  "kdor = lift_value_binary vtor"

definition kdplus where
  "kdplus = lift_value_binary vtplus"

definition kdplus_safe where
  "kdplus_safe = lift_value_binary vtplus_safe"
```

```
definition kdminus where
  "kdminus = lift_value_binary vtminus"

definition kdminus_safe where
  "kdminus_safe = lift_value_binary vtminus_safe"

definition kdmult where
  "kdmult = lift_value_binary vtmult"

definition kdmult_safe where
  "kdmult_safe = lift_value_binary vtmult_safe"

definition kdmod where
  "kdmod = lift_value_binary vtmod"

definition kdbytes_index where
  "kdbytes_index = lift_value_binary vtbytes_index"

definition kdbytes_and where
  "kdbytes_and = lift_value_binary vtbytes_and"

definition kdbytes_or where
  "kdbytes_or = lift_value_binary vtbytes_or"

definition kdbytes_xor where
  "kdbytes_xor = lift_value_binary vtbytes_xor"

definition kdbytes_not where
  "kdbytes_not = lift_value_unary vtbytes_not"

definition kdbytes_cast where
  "kdbytes_cast m = lift_value_unary (vtbytes_cast m)"

type_synonym 'a expression_monad = "('a rvalue, ex, 'a state) state_monad"

definition newStack::"'a::address expression_monad" where
  "newStack = update (λs. (Empty, s(|Stack:=fmempty|)))"

definition newMemory::"'a::address expression_monad" where
  "newMemory = update (λs. (Empty, s(|Memory:=[]|)))"

definition newCalldata::"'a::address expression_monad" where
  "newCalldata = update (λs. (Empty, s(|Calldata:=fmempty|)))"

fun the_value where
  "the_value (rvalue.Value x) = Some x"
| "the_value _ = None"

primrec lfold :: "('a::address) expression_monad list ⇒ (('a::address) valtype list, ex,('a::address)
state) state_monad"
  where
    "lfold [] = return []"
  | "lfold (m#ms) =
      do {
        l ← m;
        l' ← option Err (λ_. the_value l);
        ls ← lfold ms;
        return (l' # ls)
      }"
```

4.2 Constants (Solidity)

```
definition bool_monad where
```

```
  "bool_monad = return ∘ kdbool"

definition true_monad::"('a::address) expression_monad" where
  "true_monad = bool_monad True"

definition false_monad::"('a::address) expression_monad" where
  "false_monad = bool_monad False"

definition sint_monad ("(⟨sint⟩ _)" [70] 69) where
  "sint_monad = return ∘ kdSint"

definition bytes_monad where
  "bytes_monad n xs  = (if n ∉ {1..<33} ∨ n ≠ length xs then throw Err else return (kdBytes xs))"

definition address_monad where
  "address_monad = return ∘ kdAddress"

locale Contract =
  fixes this :: "'a::address"
begin

definition this_monad where
  "this_monad = address_monad this"

end

locale Method =
  fixes msg_sender :: "'a::address"
    and msg_value :: "256 word"
    and timestamp :: "256 word"
  assumes sender_neq_null: "msg_sender ≠ null"
begin

definition sender_monad  ("⟨sender⟩") where
  "sender_monad = address_monad msg_sender"

definition value_monad ("⟨value⟩") where
  "value_monad = sint_monad msg_value"

definition block_timestamp_monad where
  "block_timestamp_monad = sint_monad timestamp"

end

locale Keccak256 =
  fixes keccak256::"('a::address) rvalue ⇒ ('a::address) rvalue"
  assumes "⋀x y. keccak256 x = keccak256 y ⟹ x = y"
begin

definition keccak256_monad::"('a::address) expression_monad ⇒ ('a::address) expression_monad"
("⟨keccak256⟩") where
  "keccak256_monad m =
    do {
      v ← m;
      return (keccak256 v)
    }"

end
```

4.3 Unary Operations (Solidity)

```
definition lift_unary_monad ::"(('a::address) rvalue ⇒ ('a::address) rvalue option) ⇒ ('a::address)
expression_monad ⇒ ('a::address) expression_monad" where
  "lift_unary_monad op lm =
```

```
    do {
      lv ← lm;
      val ← option Err (K (op lv));
      return val
    }"

definition not_monad::"('a::address) expression_monad ⇒ ('a::address) expression_monad" ("⟨¬⟩ _" 65)
where
  "not_monad = lift_unary_monad kdnot"
```

4.4 Binary Operations (Solidity)

```
definition lift_op_monad::"(('a::address) rvalue ⇒ ('a::address) rvalue ⇒ ('a::address) rvalue option)
⇒ ('a::address) expression_monad ⇒ ('a::address) expression_monad ⇒ ('a::address) expression_monad"
where
  "lift_op_monad op lm rm =
    do {
      lv ← lm;
      rv ← rm;
      val ← option Err (K (op lv rv));
      return val
    }"

lemma lift_op_monad_simp1:
  assumes "execute lm s = Normal (lv, s')"
      and "execute rm s' = Exception (e, s'')"
    shows "execute (lift_op_monad op lm rm) s = Exception (e, s'')"
  ⟨proof⟩

lemma lift_op_monad_simp2:
  assumes "execute lm s = Normal (lv, s')"
      and "execute rm s' = NT"
    shows "execute (lift_op_monad op lm rm) s = NT"
  ⟨proof⟩

lemma lift_op_monad_simp3:
  assumes "execute lm s = Exception (e, s')"
    shows "execute (lift_op_monad op lm rm) s = Exception (e, s')"
  ⟨proof⟩

lemma lift_op_monad_simp4:
  assumes "execute lm s = NT"
    shows "execute (lift_op_monad op lm rm) s = NT"
  ⟨proof⟩

lemma lift_op_monad_simp5:
  assumes "execute lm s = Normal (lv, s')"
      and "execute rm s' = Normal (rv, s'')"
    shows "execute (lift_op_monad op lm rm) s = execute (option Err (K (op lv rv))) s''"
  ⟨proof⟩

definition equals_monad (infixl "⟨=⟩" 65) where
  "equals_monad = lift_op_monad kdequals"

lemma equals_monad_simp1[execute_simps]:
  assumes "execute lm s = Normal (lv, s')"
      and "execute rm s' = Exception (e, s'')"
    shows "execute (equals_monad lm rm) s = Exception (e, s'')"
  ⟨proof⟩

lemma equals_monad_simp2[execute_simps]:
  assumes "execute lm s = Normal (lv, s')"
      and "execute rm s' = NT"
    shows "execute (equals_monad lm rm) s = NT"
```

⟨*proof*⟩

lemma *equals_monad_simp3[execute_simps]:*
assumes *"execute lm s = Exception (e, s')"*
shows *"execute (equals_monad lm rm) s = Exception (e, s')"*
⟨*proof*⟩

lemma *equals_monad_simp4[execute_simps]:*
assumes *"execute lm s = NT"*
shows *"execute (equals_monad lm rm) s = NT"*
⟨*proof*⟩

lemma *equals_monad_simp5[execute_simps]:*
assumes *"execute lm s = Normal (lv, s')"*
and *"execute rm s' = Normal (rv, s'')"*
shows *"execute (equals_monad lm rm) s = execute (option Err (K (kdequals lv rv))) s''"*
⟨*proof*⟩

definition *less_monad* (**infixl** *"⟨<⟩" 65*) **where**
"less_monad = lift_op_monad kdless"

definition *and_monad* (**infixl** *"⟨∧⟩" 55*) **where**
"and_monad = lift_op_monad kdand"

definition *or_monad* (**infixl** *"⟨∨⟩" 54*) **where**
"or_monad = lift_op_monad kdor"

definition *plus_monad::"('a::address) expression_monad ⇒ ('a::address) expression_monad ⇒ ('a::address) expression_monad"* **where**
"plus_monad = lift_op_monad kdplus"

definition *plus_monad_safe::*
"('a::address) expression_monad ⇒ ('a::address) expression_monad ⇒ ('a::address) expression_monad"
(**infixl** *"⟨+⟩" 65*)
where
"plus_monad_safe = lift_op_monad kdplus_safe"

definition *minus_monad::"('a::address) expression_monad ⇒ ('a::address) expression_monad ⇒ ('a::address) expression_monad"* **where**
"minus_monad = lift_op_monad kdminus"

definition *minus_monad_safe::"('a::address) expression_monad ⇒ ('a::address) expression_monad ⇒ ('a::address) expression_monad"* (**infixl** *"⟨-⟩" 65*) **where**
"minus_monad_safe = lift_op_monad kdminus_safe"

definition *mult_monad::"('a::address) expression_monad ⇒ ('a::address) expression_monad ⇒ ('a::address) expression_monad"* **where**
"mult_monad = lift_op_monad kdmult"

definition *mult_monad_safe::"('a::address) expression_monad ⇒ ('a::address) expression_monad ⇒ ('a::address) expression_monad"* (**infixl** *"⟨*⟩" 65*) **where**
"mult_monad_safe = lift_op_monad kdmult_safe"

definition *mod_monad::"('a::address) expression_monad ⇒ ('a::address) expression_monad ⇒ ('a::address) expression_monad"* (**infixl** *"⟨%⟩" 65*) **where**
"mod_monad = lift_op_monad kdmod"

definition *bytes_index_monad* **where**
"bytes_index_monad = lift_op_monad kdbytes_index"

definition *bytes_and_monad* **where**
"bytes_and_monad = lift_op_monad kdbytes_and"

definition *bytes_or_monad* **where**

```
  "bytes_or_monad = lift_op_monad kdbytes_or"

definition bytes_xor_monad where
  "bytes_xor_monad = lift_op_monad kdbytes_xor"

definition bytes_not_monad where
  "bytes_not_monad = lift_unary_monad kdbytes_not"

definition bytes_cast_monad where
  "bytes_cast_monad m = lift_unary_monad (kdbytes_cast m)"
```

4.5 Store Lookups (Solidity)

```
definition (in Contract) storeLookup::
  "id ⇒ ('a::address) expression_monad list ⇒ ('a::address) expression_monad"
  ("(_ ~ₛ _)" [100, 100] 70)
where
  "storeLookup i es =
    do {
      is ← lfold es;
      sd ← option Err (λs. slookup is (state.Storage s this i));
      if storage_data.is_Value sd then return (rvalue.Value (storage_data.vt sd)) else return
(rvalue.Storage (Some (|Location=i, Offset= is|)))
    }"

definition (in Contract) storeArrayLength::"id ⇒ ('a::address) expression_monad list ⇒ ('a::address)
expression_monad" where
  "storeArrayLength i es =
    do {
      is ← lfold es;
      sd ← option Err (λs. slookup is (state.Storage s this i));
      storage_disjoint sd
        (K (throw Err))
        (λsa. return (rvalue.Value (Uint (of_nat (length (storage_data.ar sd))))))
        (K (throw Err))
    }"
```

4.6 Stack Lookups (Solidity)

```
definition stack_disjoint where
  "stack_disjoint i kf mf cf cp sf sp =
    do {
      k ← applyf Stack;
      case k $$ i of
        Some x ⇒
          (case x of
            kdata.Value v ⇒ kf v
          | kdata.Storage (Some p) ⇒ sf (Location p) (Offset p)
          | kdata.Storage None ⇒ sp
          | kdata.Memory l ⇒ mf l
          | kdata.Calldata (Some p) ⇒ cf (Location p) (Offset p)
          | kdata.Calldata None ⇒ cp)
      | None ⇒ throw Err
    }"

definition (in Contract) stackLookup::
  "id ⇒ ('a::address) expression_monad list ⇒ ('a::address) expression_monad"
  ("(_ ~ _)" [1000, 0] 70)
where
  "stackLookup i es =
    do {
      is ← lfold es;
      stack_disjoint i
        (λk. return (Value k))
        (λp. do {
```

```
        l ← option Err (λs. mlookup (state.Memory s) is p);
        md  ← option Err (λs. state.Memory s $ l);
        if mdata.is_Value md then return (rvalue.Value (mdata.vt md)) else return (rvalue.Memory l)
      })
      (λp xs. do {
        sd ← option Err (λs. state.Calldata s $$ p ⤜ clookup (xs@is));
        if call_data.is_Value sd then return (rvalue.Value (call_data.vt sd)) else return
(rvalue.Calldata (Some (⦇Location=p, Offset=xs@is⦈)))
      })
      (
        return (rvalue.Calldata None)
      )
      (λp xs. do {
        sd ← option Err (λs. slookup (xs@is) (state.Storage s this p));
        if storage_data.is_Value sd then return (rvalue.Value (storage_data.vt sd)) else return
(rvalue.Storage (Some (⦇Location=p, Offset=xs@is⦈)))
      })
      (
        return (rvalue.Storage None)
      )
  }"

definition (in Contract) arrayLength::"id ⇒ ('a::address) expression_monad list ⇒ ('a::address)
expression_monad" where
  "arrayLength i es =
    do {
      is ← lfold es;
      stack_disjoint i
        (K (throw Err))
        (λp. do {
          l ← option Err (λs. mlookup (state.Memory s) is p);
          md  ← option Err (λs. state.Memory s $ l);
          if mdata.is_Array md then return (rvalue.Value (Uint (of_nat (length (mdata.ar md))))) else
throw Err
        })
        (λp xs. do {
          sd ← option Err (λs. state.Calldata s $$ p ⤜ clookup (xs@is));
          if call_data.is_Array sd then return (rvalue.Value (Uint (of_nat (length (call_data.ar sd)))))
else throw Err
        })
        (throw Err)
        (λp xs. do {
          sd ← option Err (λs. slookup (xs@is) (state.Storage s this p));
          if storage_data.is_Array sd then return (rvalue.Value (Uint (of_nat (length (storage_data.ar
sd))))) else throw Err
        })
        (throw Err)
    }"
```

4.7 Skip (Solidity)

definition *skip_monad:: "('a rvalue, ex, ('a::address) state) state_monad" ("⟨skip⟩") where*
"skip_monad = return Empty"

4.8 Conditionals (Solidity)

definition *cond_monad::*
"('a::address) expression_monad ⇒ ('a::address) expression_monad ⇒ ('a::address) expression_monad
⇒ ('a::address) expression_monad"
("(IF _/ THEN _/ ELSE _)" [0, 0, 61] 61)
where
"cond_monad bm mt mf =
do {
b ← equals_monad bm true_monad;
if b = kdbool True then mt else if b = kdbool False then mf else throw Err

```
  }"
```

lemma *execute_cond_monad_normal_E:*
```
  assumes "execute (cond_monad bm mt mf) s = Normal (x, s')"
  obtains (1) s'' where "execute (equals_monad bm true_monad) s = Normal (kdbool True, s'')" and
"execute mt s'' = Normal (x, s')"
        | (2) s'' where "execute (equals_monad bm true_monad) s = Normal (kdbool False, s'')" and
"execute mf s'' = Normal (x, s')"
```
⟨*proof*⟩

lemma *execute_cond_monad_exception_E:*
```
  assumes "execute (cond_monad bm mt mf) s = Exception (x, s')"
  obtains (1) "execute (equals_monad bm true_monad) s = Exception (x, s')"
        | (2) s'' where "execute (equals_monad bm true_monad) s = Normal (kdbool True, s'')" and
"execute mt s'' = Exception (x, s')"
        | (3) s'' where "execute (equals_monad bm true_monad) s = Normal (kdbool False, s'')" and
"execute mf s'' = Exception (x, s')"
        | (4) a where "execute (equals_monad bm true_monad) s = Normal (a, s')" and "a ≠ kdbool True
∧ a ≠ kdbool False ∧ x = Err"
```
⟨*proof*⟩

lemma *execute_cond_monad_simp1[execute_simps]:*
```
  assumes "execute (equals_monad bm true_monad) s = Normal (kdbool True, s')"
  shows "execute (cond_monad bm mt mf) s = execute mt s'"
```
⟨*proof*⟩

lemma *execute_cond_monad_simp2[execute_simps]:*
```
  assumes "execute (equals_monad bm true_monad) s = Normal (kdbool False, s')"
  shows "execute (cond_monad bm mt mf) s = execute mf s'"
```
⟨*proof*⟩

lemma *execute_cond_monad_simp3[execute_simps]:*
```
  assumes "execute (equals_monad bm true_monad) s = Exception (e, s')"
  shows "execute (cond_monad bm mt mf) s = Exception (e, s')"
```
⟨*proof*⟩

lemma *execute_cond_monad_simp4[execute_simps]:*
```
  assumes "execute (equals_monad bm true_monad) s = NT"
  shows "execute (cond_monad bm mt mf) s = NT"
```
⟨*proof*⟩

4.9 Require/Assert (Solidity)

definition *require_monad::"('a::address) expression_monad ⇒ ('a::address) expression_monad"* **where**
```
  "require_monad em =
    do {
      e ← em;
      if e = kdbool True then return Empty else throw Err
    }"
```

definition *assert_monad :: "('a::address) expression_monad ⇒ ('a::address) expression_monad"*
("⟨assert⟩") where
```
 "assert_monad bm =
    cond_monad bm (return Empty) (throw Err)"
```

4.10 Stack Assign (Solidity)

definition *my_update_monad:: "((('a::address) state ⇒ 'b) ⇒ (('c ⇒ 'd) ⇒ ('a::address) state ⇒*
('a::address) state) ⇒ ('b ⇒ 'd option) ⇒ 'a expression_monad" **where**
```
  "my_update_monad s u f = option Err (λs'. f (s s')) ⤜ modify ∘ u ∘ K ⤜ K (return Empty)"
```

definition *memory_update_monad:: "(('a::address valtype) memory ⇒ ('a::address valtype) memory option)*
⇒ 'a expression_monad" **where**
```
  "memory_update_monad = my_update_monad state.Memory Memory_update"
```

```
context Contract
begin

definition storage_update:: "id ⇒ ('a::address valtype) storage_data ⇒ ('a::address) state ⇒
('a::address) state" where
  "storage_update i d s = s(|Storage := (state.Storage s) (this := (state.Storage s this) (i := d))|)"

definition storage_update_monad where
  "storage_update_monad xs is sd p = option Err (λs. updateStore (xs @ is) sd (state.Storage s this p))
>>= modify ∘ (storage_update p) >>= K (return Empty)"

end

definition option_disjoint where
  "option_disjoint f m = option Err f >>= m"

fun (in Contract) assign_stack::
  "id ⇒ ('a::address) valtype list ⇒ ('a::address) rvalue ⇒ ('a::address) expression_monad"
where
  "assign_stack i is (rvalue.Value v) =
    stack_disjoint i
      (K ((modify (stack_update i (kdata.Value v))) >>= K (return Empty)))
      (λp. (memory_update_monad (λm. mupdate is (p, (mdata.Value v), m))))
      (K (K (throw Err)))
      (throw Err)
      (λp xs. storage_update_monad xs is (K (storage_data.Value v)) p)
      (throw Err)"
| "assign_stack i is (rvalue.Memory p) =
    stack_disjoint i
      (K (throw Err))
      (λp'. case_list is
        (modify (stack_update i (kdata.Memory p))>>= K (return Empty))
        (K (K (memory_update_monad (λm. (m$p) >>= (λv. mupdate is (p', v, m)))))))
      (K (K (throw Err)))
      (throw Err)
      (λp' xs. option_disjoint
        (λs. read_storage (state.Memory s) p)
        (λsd. storage_update_monad xs is (K sd) p'))
      (throw Err)"
| "assign_stack i is (rvalue.Calldata (Some (|Location=p, Offset=xs|))) =
    stack_disjoint i
      (K (throw Err))
      (λp'. option_disjoint
        (λs. state.Calldata s $$ p >>= clookup xs)
        (λcd. memory_update_monad (mupdate is ∘ (read_calldata_memory cd p'))))
      (K (K (throw Err)))
      (modify (stack_update i (kdata.Calldata (Some (|Location=p, Offset= xs|)))) >>= K (return Empty))
      (λp' xs'. option_disjoint
        (λs. state.Calldata s $$ p >>= clookup (xs @ is))
        (λcd. storage_update_monad xs' is (K (read_calldata_storage cd)) p'))
      (throw Err)"
| "assign_stack i is (rvalue.Calldata None) = throw Err"
| "assign_stack i is (rvalue.Storage (Some (|Location=p, Offset=xs|))) =
    stack_disjoint i
      (K (throw Err))
      (λp'. option_disjoint
        (λs. slookup xs (state.Storage s this p))
        (λsd. memory_update_monad
          (λm. read_storage_memory sd p' m >>=
            mupdate is)))
      (K (K (throw Err)))
      (throw Err)
      (λp' xs'. case_list is
        (modify (stack_update i (kdata.Storage (Some (|Location=p, Offset= xs|)))) >>= K (return Empty))
```

```
        (K (K (option_disjoint
          (λs. slookup (xs @ is) (state.Storage s this p))
          (λsd. storage_update_monad xs' [] (K sd) p')))))
      (modify (stack_update i (kdata.Storage (Some (|Location=p, Offset= xs|)))) ≫= K (return Empty))"
| "assign_stack i is (rvalue.Storage None) = throw Err"
| "assign_stack i is rvalue.Empty = throw Err"

definition (in Contract) assign_stack_monad::
  "String.literal ⇒ ('a rvalue, ex, 'a state) state_monad list ⇒ ('a rvalue, ex, 'a state)
state_monad ⇒ ('a rvalue, ex, 'a state) state_monad"
  ("(_ _ ::= _)" [1000, 61, 0] 61)
where
  "assign_stack_monad i es m ≡
    do {
      val ← m;
      is ← lfold es;
      assign_stack i is val;
      return Empty
    }"
```

4.11 Storage Assignment (Solidity)

```
fun (in Contract) assign_storage:: "id ⇒ ('a::address) valtype list ⇒ ('a::address) rvalue ⇒
('a::address) expression_monad" where
  "assign_storage i is (rvalue.Value v) = storage_update_monad [] is (K (storage_data.Value v)) i"
| "assign_storage i is (rvalue.Memory p) =
    (option_disjoint
      (λs. read_storage (state.Memory s) p)
      (λsd. storage_update_monad [] is (K sd) i))"
| "assign_storage i is (rvalue.Calldata (Some (|Location=p, Offset=xs|))) =
    (option_disjoint
      (λs. state.Calldata s $$ p ≫= clookup xs)
      (λcd. storage_update_monad [] is (K (read_calldata_storage cd)) i))"
| "assign_storage i is (rvalue.Calldata None) = throw Err"
| "assign_storage i is (rvalue.Storage (Some (|Location=p, Offset=xs|))) =
    (option_disjoint
      (λs. slookup xs (state.Storage s this p))
      (λsd. storage_update_monad [] is (K sd) i))"
| "assign_storage i is (rvalue.Storage None) = throw Err"
| "assign_storage i is rvalue.Empty = throw Err"

definition (in Contract) assign_storage_monad
("(_ _ ::=ₛ _)" [61, 62, 61] 60)
  where
  "assign_storage_monad i es m ≡
    do {
      val ← m;
      is ← lfold es;
      assign_storage i is val
    }"
```

4.12 Loops (Solidity)

```
lemma true_monad_mono[mono]: "mono_sm (λ_. true_monad)"
  ⟨proof⟩

lemma cond_K [partial_function_mono]: "mono_sm (λf. K (f x) y)"
⟨proof⟩

lemma lift_op_monad_mono:
  assumes "mono_sm A" and "mono_sm B"
  shows "mono_sm (λf. lift_op_monad op (A f) (B f))"
  ⟨proof⟩

lemma equals_monad_mono[mono]:
```

assumes *"mono_sm A"* **and** *"mono_sm B"*
shows *"mono_sm (λf. equals_monad (A f) (B f))"*
⟨*proof*⟩

lemma *cond_mono [partial_function_mono, mono]:*
assumes *"mono_sm A"* **and** *"mono_sm B"* **and** *"mono_sm C"*
shows *"mono_sm (λf. cond_monad (A f) (B f) (C f))"*
⟨*proof*⟩

partial_function *(sm) while_monad :: "('a::address) expression_monad ⇒ ('a::address) expression_monad ⇒ ('a::address) expression_monad"* **where**
"while_monad c m = cond_monad c (bind m (K (while_monad c m))) (return Empty)"

The partial function package provides us with three properties:

- A simplifier rule *while_monad ?c ?m = IF ?c THEN ?m ≫= K (while_monad ?c ?m) ELSE return Empty*
- A general induction rule ⟦*sm.admissible (λwhile_monad. ?P (curry while_monad)); ?P (λwhile_monad c. empty_result); ⋀f. ?P f ⟹ ?P (λa b. IF a THEN b ≫= K (f a b) ELSE return Empty)*⟧ ⟹ *?P Solidity.while_monad*
- An induction rule for partial correctness ⟦⋀*while_monad c m ca ma. ⟦⋀a b h r. effect (while_monad a b) h r ⟹ ?P a b h r; effect (IF c THEN m ≫= K (while_monad c m) ELSE return Empty) ca ma*⟧ ⟹ *?P c m ca ma; effect (Solidity.while_monad ?c ?m) ?h ?r*⟧ ⟹ *?P ?c ?m ?h ?r*

4.13 Internal Method Calls (Solidity)

definition *icall* **where**
```
"icall m =
do {
  x ← applyf Stack;
  r ← m;
  modify (Stack_update (K x));
  return r
}"
```

lemma *icall_mono[mono]:*
assumes *"mono_sm (λx. m x)"*
shows *"mono_sm (λx. icall (m x))"*
⟨*proof*⟩

4.14 External Method Calls (Solidity)

definition *ecall* **where**
```
"ecall m =
do {
  s ← get;
  r ← m;
  modify (λs'. s'(|Stack := state.Stack s, Memory := state.Memory s, Calldata := state.Calldata s|));
  return r
}"
```

lemma *ecall_mono[mono]:*
assumes *"mono_sm (λx. m x)"*
shows *"mono_sm (λx. ecall (m x))"*
⟨*proof*⟩

4.15 Transfer (Solidity)

fun *readValue:: "('a::address) rvalue ⇒ ((('a::address) valtype, ex, ('a::address) state) state_monad)"* **where**
```
  "readValue (rvalue.Value x) = return x"
| "readValue _ = throw Err"
```

```
fun readAddress:: "('a::address) valtype ⇒ (((('a::address), ex, ('a::address) state) state_monad)"
where
  "readAddress (Address x) = return x"
| "readAddress _ = throw Err"

fun readSint:: "('a::address) valtype ⇒ ((256 word, ex, ('a::address) state) state_monad)" where
  "readSint (Uint x) = return x"
| "readSint _ = throw Err"

context Contract
begin

abbreviation balance_update:: "nat ⇒ ('a::address) state ⇒ ('a::address) state" where
  "balance_update ≡ balances_update this"

definition inv:: "'a rvalue × ('a::address) state + ex × ('a::address) state ⇒ (('a::address) state
⇒ bool) ⇒ (('a::address) state ⇒ bool) ⇒ bool" where
  "inv r P Q ≡ (case r of Inl a ⇒ P (snd a)
                        | Inr a ⇒ Q (snd a))"

definition inv_state:: "((id ⇒ ('a::address valtype) storage_data) × nat ⇒ bool) ⇒ ('a::address)
state ⇒ bool"
  where "inv_state i s = i (state.Storage s this, state.Balances s this)"

definition post:: "('a::address) state ⇒ 'a rvalue × ('a::address) state + ex × ('a::address) state
⇒ ((String.literal ⇒ 'a valtype storage_data) × nat ⇒ bool) ⇒ ((String.literal ⇒ 'a  valtype
storage_data) × nat ⇒ bool) ⇒ (('a::address) state ⇒ ('a::address) rvalue ⇒ ('a::address) state ⇒
bool) ⇒ bool" where
  "post s r I_s I_e P ≡ (case r of Inl a ⇒ P s (fst a) (snd a) ∧ inv_state I_s (snd a)
                              | Inr a ⇒ inv_state I_e (snd a))"


lemma post_exc_true:
  assumes "effect (exc x) s r"
      and "⋀r. effect x s r ⟹ post s r I (K True) P"
    shows "post s r I (K True) P"
  ⟨proof⟩

lemma post_exc_false:
  assumes "effect (exc x) s r"
      and "⋀r. effect x s r ⟹ post s r I (K False) P"
    shows "post s r I (K False) P"
  ⟨proof⟩


lemma post_true:
  assumes "effect (exc x) s r"
      and "inv_state I s"
      and "post s r I (K True) P"
    shows "post s r I I P"
  ⟨proof⟩

end

locale External = Contract +
  constrains this :: "'a::address"
  fixes external::"('d ⇒ 'a expression_monad) ⇒ ('a::address) expression_monad"
  assumes external_mono[mono]: "mono_sm (λcall. external call)"
begin


definition transfer_monad::
  "('d ⇒ 'a expression_monad) ⇒ ('a::address) expression_monad ⇒ ('a::address) expression_monad ⇒
('a::address) expression_monad"
```

```
("⟨transfer⟩")
where
  "transfer_monad call am vm =
    do {
      ak ← am;
      av ← readValue ak;
      a  ← readAddress av;
      vk ← vm;
      vv ← readValue vk;
      v  ← readSint vv;
      assert Err (λs. Balances s this ≥ unat v);
      modify (λs. balances_update this (Balances s this - unat v) s);
      modify (λs. balances_update a (Balances s a + unat v) s);
      ecall (external call)
    }"

lemma transfer_mono[mono]:
  shows "monotone sm.le_fun sm_ord
     (λf. transfer_monad f m n)"
  ⟨proof⟩

end
```

4.16 Solidity (Solidity)

```
locale Solidity = Keccak256 + Method + External +
  constrains keccak256::"('a::address) rvalue ⇒ ('a::address) rvalue"
         and msg_sender :: "'a::address"
         and this::"'a::address"
         and external::"('d ⇒ 'a expression_monad) ⇒ ('a::address) expression_monad"
begin
  definition init_balance:: "('a rvalue, ex, ('a::address) state) state_monad" where
    "init_balance = modify (λs. balance_update (Balances s this + unat msg_value) s) ≫= K (return
Empty)"

  definition init_balance_np:: "('a rvalue, ex, ('a::address) state) state_monad" where
    "init_balance_np = modify (λs. balance_update (Balances s this) s) ≫= K (return Empty)"

end
```

4.17 Arrays (Solidity)

```
definition array where "array i x = replicate i x"

lemma length_array[simp]: "length (array x y) = x"
  ⟨proof⟩


lemma fold_map_write_replicate_length:
  assumes "fold_map Memory.write (replicate n (adata.Value v)) m = (x1, x2)"
    shows "length x1 = n"
  ⟨proof⟩

lemma fold_map_write_replicate_value:
  assumes "fold_map Memory.write (replicate n (adata.Value (Uint 0))) m = (x1, x2)"
      and "x < n"
    shows "x1 ! x < length x2 ∧ (∃ix. x2 ! (x1 ! x) = mdata.Value (Uint ix))"
  ⟨proof⟩

lemma write_array_typing_value:
  assumes "Memory.write (adata.Array (array (unat si) (adata.Value (Uint 0)))) [] = (x1, x2)"
    shows "x1<length x2 ∧ (∃ma0. x2 ! x1 = mdata.Array ma0 ∧ (∀i<length ma0. (ma0 ! i) < length x2 ∧
(∃ix. x2 ! (ma0 ! i) = mdata.Value (Uint ix))))"
⟨proof⟩
```

lemma *mupdate_array_typing_value:*
  assumes *"state.Memory sa ! ml = mdata.Array ma0"*
      and *"∀i<length ma0. (ma0 ! i) < length (state.Memory sa) ∧ (∃ix. state.Memory sa ! (ma0 ! i) = mdata.Value (Uint ix))"*
      and *"mupdate [Uint xa] (ml, mdata.Value (Uint x), state.Memory sa) = Some yg"*
    shows *"∃ma0. yg ! ml = mdata.Array ma0*
        ∧ (∀i<length ma0. (ma0 ! i) < length yg ∧ (∃ix. yg ! (ma0 ! i) = mdata.Value (Uint ix)))"
⟨*proof*⟩

4.18 Declarations (Solidity)

definition (**in** *Contract*) *initStorage::"id ⇒ ('a::address valtype) storage_data ⇒ ('a::address) expression_monad"* **where**
  "initStorage i v ≡ modify (λs. storage_update i v s) ≫= K (return Empty)"

definition *kinit::"('a::address valtype) kdata ⇒ id ⇒ ('a::address) expression_monad"* **where**
  "kinit v i ≡ modify (stack_update i v) ≫= K (return Empty)"

definition *init::"('a::address) valtype ⇒ id ⇒ ('a::address) expression_monad"* **where**
  "init ≡ kinit ∘ kdata.Value"

definition *"write"::"('a::address valtype) adata ⇒ id ⇒ ('a::address) expression_monad"* **where**
  "write c i ≡ update (λs. let (l,m) = Memory.write c (state.Memory s) in (Empty, s(|Stack := fmupd i (kdata.Memory l) (Stack s), Memory := m|)))"

definition *cinit::"('a::address valtype) call_data ⇒ id ⇒ ('a::address) expression_monad"* **where**
  "cinit c i ≡ modify (calldata_update i c ∘ stack_update i (kdata.Calldata (Some (|Location=i,Offset=[]|)))) ≫= K (return Empty)"

4.18.1 Stack Variables

datatype *VType =*
  TBool | TSint | TAddress | TBytes nat — width should be restricted to [1..32]

4.18.2 Default values

definition *mapping* **where** *"mapping x = (λ_. x)"*

fun *default:: "VType ⇒ 'a::address valtype"* **where**
   "default TBool = Bool False"
 | "default TSint = Uint 0"
 | "default TAddress = Address null"
 | "default (TBytes n) = Bytes (array n (CHR 0x00))"

definition *decl::"VType ⇒ id ⇒ ('a::address) expression_monad"*
where
  "decl ≡ init ∘ default"

abbreviation *decl'::"id ⇒ VType ⇒ ('a::address) expression_monad"*
  ("(_ :: _)" [61, 61] 60)
where
  "decl' x y ≡ decl y x"

4.18.3 Storage Variables

datatype *SType =*
  TValue VType | TArray nat SType | DArray SType | TMap SType SType | TEnum "SType list"

term *"DArray (TValue (TBytes 1))"*
⟨*ML*⟩

fun *sdefault:: "SType ⇒ 'a::address valtype storage_data"* **where**
   "sdefault (TValue t) = storage_data.Value (default t)"
 | "sdefault (TArray l t) = storage_data.Array (array l (sdefault t))"
 | "sdefault (DArray t) = storage_data.Array []"

```
 | "sdefault (TMap _ t) = storage_data.Map (mapping (sdefault t))"
 | "sdefault (TEnum xs) = storage_data.Array []"

definition sinit::"id ⇒ ('a::address) expression_monad" where
  "sinit i ≡ modify (stack_update i (kdata.Storage None)) ≫= K (return Empty)"


fun sdecl::"SType ⇒ id ⇒ ('a::address) expression_monad" where
   "sdecl (TValue _) = K (throw Err)"
 | "sdecl _ = sinit"
declare sdecl.simps[simp del]

fun push where
  "push d (storage_data.Array xs) = Some (storage_data.Array (xs @@ d))"
| "push _ _ = None"

definition (in Contract) allocate::"id ⇒ ('a::address) expression_monad list ⇒ ('a::address valtype)
storage_data ⇒ ('a::address) expression_monad" where
  "allocate i es d =
    do {
      is ← lfold es;
      ar ← option Err (λs. slookup is (state.Storage s this i) ≫= push d);
      storage_update_monad [] is (K ar) i
    }"
```

4.18.4 Calldata Variables

```
datatype CType =
  TValue VType | TArray nat CType | DArray CType | TEnum "CType list"

fun cdefault:: "CType ⇒ 'a::address valtype adata" where
   "cdefault (TValue t) = adata.Value (default t)"
 | "cdefault (TArray l t) = adata.Array (array l (cdefault t))"
 | "cdefault (DArray t) = adata.Array []"
 | "cdefault (TEnum xs) = adata.Array []"
```

4.18.5 Memory Variables

```
definition mdecl::"CType ⇒ id ⇒ ('a::address) expression_monad" where
  "mdecl = write ∘ cdefault"

definition create_memory_array where
  "create_memory_array i t sm =
    do {
      s ← sm;
      (case s of
        rvalue.Value (Uint s') ⇒ write (adata.Array (array (unat s') (cdefault t))) i
      | _ ⇒ throw Err)
    }"

end
```

5 ML Commands

In this chapter, we present the implementation of the Isabelle commands to specify a contract and verify invariants.

```
theory Contract
  imports Solidity WP
  keywords "contract" :: thy_decl
       and "constructor"
       and "cfunction"
       and "external"
       and "memory"
       and "param"
       and "calldata"
       and "payable"
       and "verification" :: thy_goal
       and "strong"
       and "invariant"::thy_decl

begin
⟨ML⟩

end
```

6 Weakest Precondition Calculus

In this chapter, we present a weakest precondition calculus and corresponding verification condition generator.
theory *WP*
imports *Solidity "HOL-Eisbach.Eisbach"*
begin

6.1 Weakest precondition calculus (WP)

named_theorems *wprules*
named_theorems *wperules*
named_theorems *wpdrules*
named_theorems *wpsimps*

declare(**in** *Contract*) *inv_state_def[wpsimps]*
declare *icall_def[wpsimps]*
declare *ecall_def[wpsimps]*

method *wp* **declares** *wprules wpdrules wperules wpsimps = (rule wprules | drule wpdrules | erule wperules | simp add: wpsimps)*
method *vcg* **declares** *wprules wpdrules wperules wpsimps = wp+*

6.1.1 Simplification rules

lemma *mapping[wpsimps]:*
"mapping x y = x"
⟨*proof*⟩

lemma *Value_vt[wpsimps]:*
assumes *"storage_data.Value x = v"*
shows *"storage_data.vt v = x"*
⟨*proof*⟩

Kdata

lemma *kdbool_simp[wpsimps]:*
"kdbool x = Value (Bool x)"
⟨*proof*⟩

lemma *kdSint_simp[wpsimps]:*
"kdSint x = Value (Uint x)"
⟨*proof*⟩

lemma *kdBytes_simp[wpsimps]:*
"kdBytes xs = Value (Bytes xs)"
⟨*proof*⟩

lemma *kdAddress_simp[wpsimps]:*
"kdAddress x = Value (Address x)"
⟨*proof*⟩

lemma *kdminus[wpsimps]:*
"kdminus (rvalue.Value (Uint l)) (rvalue.Value (Uint r)) = Some (rvalue.Value (Uint (l - r)))"
⟨*proof*⟩

lemma *kdminus_safe[wpsimps]:*
assumes *"r ≤ l"*
shows *"kdminus_safe (rvalue.Value (Uint l)) (rvalue.Value (Uint r)) = Some (rvalue.Value (Uint (l - r)))"*

⟨*proof*⟩

lemma *kdminus_safe_dest[wpdrules]:*
assumes *"kdminus_safe (rvalue.Value (Uint l)) (rvalue.Value (Uint r)) = Some ya"*
shows *"r ≤ l ∧ ya = rvalue.Value (Uint (l - r))"*
⟨*proof*⟩

lemma *kdminus_storage[wpsimps]:*
"kdminus (rvalue.Storage x) z = None"
⟨*proof*⟩

lemma *kdplus[wpsimps]:*
"kdplus (rvalue.Value (Uint l)) (rvalue.Value (Uint r)) = Some (rvalue.Value (Uint (l + r)))"
⟨*proof*⟩

lemma *kdplus_safe[wpsimps]:*
assumes *"unat l + unat r < 2^256"*
shows *"kdplus_safe (rvalue.Value (Uint l)) (rvalue.Value (Uint r)) = Some (rvalue.Value (Uint (l + r)))"*
⟨*proof*⟩

lemma *kdplus_safe_dest[wpdrules]:*
assumes *"kdplus_safe (rvalue.Value (Uint l)) (rvalue.Value (Uint r)) = Some ya"*
shows *"unat l + unat r < 2^256 ∧ ya = rvalue.Value (Uint (l + r))"*
⟨*proof*⟩

lemma *kdmult[wpsimps]:*
*"kdmult (rvalue.Value (Uint l)) (rvalue.Value (Uint r)) = Some (rvalue.Value (Uint (l * r)))"*
⟨*proof*⟩

lemma *kdmult_safe[wpsimps]:*
assumes *"unat l * unat r < 2^256"*
shows *"kdmult_safe (rvalue.Value (Uint l)) (rvalue.Value (Uint r)) = Some (rvalue.Value (Uint (l * r)))"*
⟨*proof*⟩

lemma *kdmult_safe_dest[wpdrules]:*
assumes *"kdmult_safe (rvalue.Value (Uint l)) (rvalue.Value (Uint r)) = Some ya"*
shows *"unat l * unat r < 2^256 ∧ ya = rvalue.Value (Uint (l * r))"*
⟨*proof*⟩

Updates

lemma *stack_stack_update_diff[wpsimps]:*
assumes *"i ≠ i'"*
shows *"Stack (stack_update i x s) $$ i' = Stack s $$ i'"*
⟨*proof*⟩

lemma (**in** *Contract*) *stack_storage_update[wpsimps]:*
"Stack (storage_update i x s) = Stack s"
⟨*proof*⟩

lemma *stack_balances_update[wpsimps]:*
"Stack (balances_update i x s) = Stack s"
⟨*proof*⟩

lemma *stack_calldata_update[wpsimps]:*
"Stack (calldata_update i x s) = Stack s"
⟨*proof*⟩

lemma *stack_update_eq[wpsimps]:*
"Stack (stack_update i x s) $$ i = Some x"
⟨*proof*⟩

lemma *memory_balances_update[wpsimps]:*
"state.Memory (balances_update i x s) = state.Memory s"
⟨*proof*⟩

lemma *memory_stack_update[wpsimps]:*
"state.Memory (stack_update i x s) = state.Memory s"
⟨*proof*⟩

lemma *calldata_balances_update[wpsimps]:*
"state.Calldata (balances_update i x s) = state.Calldata s"
⟨*proof*⟩

lemma *calldata_stack_update[wpsimps]:*
"state.Calldata (stack_update i x s) = state.Calldata s"
⟨*proof*⟩

lemma *storage_stack_update[wpsimps]:*
"state.Storage (stack_update i v s) = state.Storage s"
⟨*proof*⟩

lemma *storage_calldata_update[wpsimps]:*
"state.Storage (calldata_update i v s) = state.Storage s"
⟨*proof*⟩

lemma *storage_balances_update[wpsimps]:*
"state.Storage (balances_update i v s) = state.Storage s"
⟨*proof*⟩

lemma *calldata_calldata_update[wpsimps]:*
"state.Calldata (calldata_update i v s) $$ i = Some v"
⟨*proof*⟩

lemma (**in** *Contract*) *storage_update_diff[wpsimps]:*
assumes *"i ≠ i'"*
shows *"state.Storage (storage_update i x s) this i' = state.Storage s this i'"*
⟨*proof*⟩

lemma (**in** *Contract*) *storage_update_eq[wpsimps]:*
"state.Storage (storage_update i x s) this i = x"
⟨*proof*⟩

lemma (**in** *Contract*) *balances_storage_update[wpsimps]:*
"Balances (storage_update i' x s) = Balances s"
⟨*proof*⟩

lemma *balances_stack_update[wpsimps]:*
"Balances (stack_update i' x s) = Balances s"
⟨*proof*⟩

lemma *balances_balances_update_diff[wpsimps]:*
assumes *"i ≠ i'"*
shows *"Balances (balances_update i x s) i' = Balances s i'"*
⟨*proof*⟩

lemma *balances_balances_update_same[wpsimps]:*
"Balances (balances_update i x s) i = x"
⟨*proof*⟩

6.1.2 Destruction rules

lemma *some_some[wpdrules]:*
assumes *"Some x = Some y"*
shows *"x = y"* ⟨*proof*⟩

6.1.3 Weakest Precondition

definition *wp::"('a, 'b, 'c) state_monad ⇒ ('a ⇒ 'c ⇒ bool) ⇒ ('b ⇒ 'c ⇒ bool) ⇒ 'c ⇒ bool"*
where
"wp f P E s ≡
(case execute f s of
Normal (r,s') ⇒ P r s'
| Exception (e,s') ⇒ E e s'
| NT ⇒ True)"

lemma *wpI:*
assumes *"⋀r s'. execute f s = Normal (r, s') ⟹ P r s'"*
and *"⋀e s'. execute f s = Exception (e, s') ⟹ E e s'"*
shows *"wp f P E s"*
⟨*proof*⟩

lemma *wpE:*
assumes *"wp f P E s"*
obtains *(1) r s'* **where** *"execute f s = Normal (r, s') ∧ P r s'"*
| (2) e s' **where** *"execute f s = Exception (e, s') ∧ E e s'"*
| (3) "execute f s = NT"
⟨*proof*⟩

lemma *wp_simp1:*
assumes *"execute f s = Normal (r, s')"*
shows *"wp f P E s = P r s'"*
⟨*proof*⟩

lemma *wp_simp2:*
assumes *"execute f s = Exception (e, s')"*
shows *"wp f P E s = E e s'"*
⟨*proof*⟩

lemma *wp_simp3:*
assumes *"execute f s = NT"*
shows *"wp f P E s"*
⟨*proof*⟩

lemma *wp_if[wprules]:*
assumes *"b ⟹ wp a P E s"*
and *"¬ b ⟹ wp c P E s"*
shows *"wp (if b then a else c) P E s"*
⟨*proof*⟩

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

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

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

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

lemma *wp_lfold:*
assumes *"P [] s"*
assumes *"⋀a list. xs = a#list ⟹ wp (a ⤜ (λl. option Err (λ_. the_value l) ⤜ (λl'. lfold list ⤜ (λls. return (l' # ls))))) P E s"*
shows *"wp (lfold xs) P E s"*
⟨*proof*⟩

lemma *result_cases2[cases type: result]:*
fixes *x :: "('a × 's, 'e × 's) result"*
obtains *(n) a s e* **where** *"x = Normal (a, s) ∨ x = Exception (e, s)"*
| *(t) "x = NT"*
⟨*proof*⟩

lemma *wpmodify[wprules]:*
assumes *"P () (f s)"*
shows *"wp (modify f) P E s"*
⟨*proof*⟩

lemma *wpnewStack[wprules]:*
assumes *"P Empty (s(|Stack := {$$}|))"*
shows *"wp newStack P E s"*
⟨*proof*⟩

lemma *wpnewMemory[wprules]:*
assumes *"P Empty (s(|Memory := []|))"*
shows *"wp newMemory P E s"*
⟨*proof*⟩

lemma *wpnewCalldata[wprules]:*
assumes *"P Empty (s(|Calldata := {$$}|))"*
shows *"wp newCalldata P E s"*
⟨*proof*⟩

lemma *wp_lift_op_monad:*
assumes *"wp lm (λa. wp (rm ⤜ (λrv. option Err (K (op a rv)) ⤜ return)) P E) E s"*
shows *"wp (lift_op_monad op lm rm) P E s"*
⟨*proof*⟩

lemma *wp_equals_monad[wprules]:*
assumes *"wp lm (λa. wp (rm ⤜ (λrv. option Err (K (kdequals a rv)) ⤜ return)) P E) E s"*
shows *"wp (equals_monad lm rm) P E s"*
⟨*proof*⟩

lemma *wp_less_monad[wprules]:*
assumes *"wp lm (λa. wp (rm ⤜ (λrv. option Err (K (kdless a rv)) ⤜ return)) P E) E s"*
shows *"wp (less_monad lm rm) P E s"*
⟨*proof*⟩

lemma *wp_mod_monad[wprules]:*
assumes *"wp lm (λa. wp (rm ⤜ (λrv. option Err (K (kdmod a rv)) ⤜ return)) P E) E s"*
shows *"wp (mod_monad lm rm) P E s"*
⟨*proof*⟩

lemma *wp_minus_monad[wprules]:*
assumes *"wp lm (λa. wp (rm ⤜ (λrv. option Err (K (kdminus a rv)) ⤜ return)) P E) E s"*
shows *"wp (minus_monad lm rm) P E s"*
⟨*proof*⟩

lemma *wp_minus_monad_safe[wprules]:*
assumes *" wp lm (λa. wp (rm ⤜ (λrv. option Err (K (kdminus_safe a rv)) ⤜ return)) P E) E s"*
shows *"wp (minus_monad_safe lm rm) P E s"*
⟨*proof*⟩

lemma *wp_plus_monad[wprules]:*

assumes *"wp lm (λa. wp (rm ≫= (λrv. option Err (K (kdplus a rv)) ≫= return)) P E) E s"*
shows *"wp (plus_monad lm rm) P E s"*
⟨*proof*⟩

lemma *wp_plus_monad_safe[wprules]:*
assumes *"wp lm (λa. wp (rm ≫= (λrv. option Err (K (kdplus_safe a rv)) ≫= return)) P E) E s"*
shows *"wp (plus_monad_safe lm rm) P E s"*
⟨*proof*⟩

lemma *wp_mult_monad[wprules]:*
assumes *"wp lm (λa. wp (rm ≫= (λrv. option Err (K (kdmult a rv)) ≫= return)) P E) E s"*
shows *"wp (mult_monad lm rm) P E s"*
⟨*proof*⟩

lemma *wp_mult_monad_safe[wprules]:*
assumes *"wp lm (λa. wp (rm ≫= (λrv. option Err (K (kdmult_safe a rv)) ≫= return)) P E) E s"*
shows *"wp (mult_monad_safe lm rm) P E s"*
⟨*proof*⟩

lemma *wp_bool_monad[wprules]:*
assumes *"P (kdbool b) s"*
shows *"wp (bool_monad b) P E s"*
⟨*proof*⟩

lemma *wp_true_monad[wprules]:*
assumes *"P (kdbool True) s"*
shows *"wp true_monad P E s"*
⟨*proof*⟩

lemma *wp_false_monad[wprules]:*
assumes *"P (kdbool False) s"*
shows *"wp false_monad P E s"*
⟨*proof*⟩

lemma *wp_or_monad[wprules]:*
assumes *"wp l (λa. wp (r ≫= (λrv. option Err (K (lift_value_binary vtor a rv)) ≫= return)) P E) E s"*
shows *"wp (or_monad l r) P E s"*
⟨*proof*⟩

lemma *wp_sint_monad[wprules]:*
assumes *"P (kdSint x) s"*
shows *"wp (sint_monad x) P E s"*
⟨*proof*⟩

lemma *wp_bytest_monad[wprules]:*
assumes *"P (kdBytes x) s" "n = length x" "n ∈ {1..<33}"*
shows *"wp (bytes_monad n x) P E s"*
⟨*proof*⟩

lemma (**in** *Method*) *wp_value_monad[wprules]:*
assumes *"P (kdSint msg_value) s"*
shows *"wp value_monad P E s"*
⟨*proof*⟩

lemma (**in** *Method*) *wp_stamp_monad[wprules]:*
assumes *"P (kdSint timestamp) s"*
shows *"wp block_timestamp_monad P E s"*
⟨*proof*⟩

lemma *wp_cond_monad[wprules]:*
assumes *"wp bm (λa. wp (true_monad ≫= (λrv. option Err (K (kdequals a rv)) ≫= return)) (λa. wp (if a = kdbool True then mt else if a = kdbool False then fm else throw Err) P E) E) E s"*
shows *"wp (cond_monad bm mt fm) P E s"*

⟨*proof*⟩

lemma *wp_assert_monad[wprules]:*
assumes *"wp (Solidity.cond_monad bm (return Empty) (throw Err)) P E s"*
shows *"wp (assert_monad bm) P E s"*
⟨*proof*⟩

lemma *wpoption[wprules]:*
assumes *"⋀y. f s = Some y ⟹ P y s"*
and *"f s = None ⟹ E x s"*
shows *"wp (option x f) P E s"*
⟨*proof*⟩

lemma *wp_lift_unary_monad:*
assumes *"wp lm (λa. wp (option Err (K (op a)) ⤜ return) P E) E s"*
shows *"wp (lift_unary_monad op lm) P E s"*
⟨*proof*⟩

lemma *wp_not_monad[wprules]:*
assumes *"wp lm (λa. wp (option Err (K (kdnot a)) ⤜ return) P E) E s"*
shows *"wp (not_monad lm) P E s"*
⟨*proof*⟩

lemma *wp_address_monad[wprules]:*
assumes *"P (kdAddress a) s"*
shows *"wp (address_monad a) P E s"*
⟨*proof*⟩

lemma(**in** *Method) wp_sender_monad[wprules]:*
assumes *"P (kdAddress msg_sender) s"*
shows *"wp sender_monad P E s"*
⟨*proof*⟩

lemma *wp_require_monad[wprules]:*
assumes *"wp (x ⤜ (λv. if v = rvalue.Value (Bool True) then return Empty else throw Err)) P E s"*
shows *"wp (require_monad x) P E s"*
⟨*proof*⟩

lemma (**in** *Contract) wp_storeLookup[wprules]:*
assumes *"wp (lfold es)*
(λa. wp (option Err (λs. slookup a (state.Storage s this i)) ⤜
(λsd. if storage_data.is_Value sd then return (rvalue.Value (storage_data.vt sd)) else return (rvalue.Storage (Some (⦇Location=i, Offset= a⦈)))))
P E)
E s"
shows *"wp (storeLookup i es) P E s"*
⟨*proof*⟩

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

lemma *wp_bool[wprules]:*
"wp (bool_monad b) (λa _. a = kdbool b) (K x) s"
⟨*proof*⟩

lemma *wpskip[wprules]:*
assumes *"P Empty s"*
shows *"wp skip_monad P E s"*
⟨*proof*⟩

lemma *effect_bind:*
assumes "effect (m ≫= (λx. n x)) ss r"
and "execute m ss = Normal (a2, s)"
shows "effect (n a2) s r"
⟨*proof*⟩

lemma *effect_cond_monad:*
assumes "effect (Solidity.cond_monad c mt mf) ss r"
and "execute (equals_monad c true_monad) ss = Normal (kdbool True, s)"
shows "effect mt s r"
⟨*proof*⟩

lemma *wpwhile:*
assumes "⋀s. iv s
⟹ wp (equals_monad c true_monad)
(λa s. (a = kdbool True ⟶ wp m (K iv) E s) ∧
(a = kdbool False ⟶ P Empty s) ∧
(a ≠ kdbool False ∧ a ≠ kdbool True ⟶ E Err s))
E s"
and "iv s"
shows "wp (while_monad c m) P E s"
⟨*proof*⟩

lemma *wp_applyf[wprules]:*
assumes "P (f s) s"
shows "wp (applyf f) P E s"
⟨*proof*⟩

lemma *wp_case_option[wprules]:*
assumes "x = None ⟹ wp a P E s"
and "⋀a. x = Some a ⟹ wp (b a) P E s"
shows "wp (case x of None ⇒ a | Some x ⇒ b x) P E s"
⟨*proof*⟩

lemma *wp_case_kdata[wprules]:*
assumes "⋀x1. a = kdata.Storage x1 ⟹ wp (S x1) P E s"
and "⋀x2. a = kdata.Memory x2 ⟹ wp (M x2) P E s"
and "⋀x3. a = kdata.Calldata x3 ⟹ wp (C x3) P E s"
and "⋀x4. a = kdata.Value x4 ⟹ wp (V x4) P E s"
shows "wp (case a of kdata.Storage p ⇒ S p | kdata.Memory l ⇒ M l | kdata.Calldata p ⇒ C p |
kdata.Value x ⇒ V x) P E s"
⟨*proof*⟩

lemma *wp_init[wprules]:*
assumes "P Empty (stack_update i (kdata.Value v) s)"
shows "wp (init v i) P E s"
⟨*proof*⟩

lemma *wp_decl[wprules]:*
assumes "wp (init (Solidity.default t) i) P E s"
shows "wp (decl t i) P E s"
⟨*proof*⟩

lemma *wp_write[wprules]:*
assumes "⋀x1 x2.
Memory.write c (state.Memory s) = (x1, x2) ⟹
P Empty (s(|Stack := Stack s(i $$:= kdata.Memory x1), Memory := x2|))"
shows "wp (write c i) P E s"
⟨*proof*⟩

lemma *wp_sinit[wprules]:*
assumes "P Empty (stack_update i (kdata.Storage None) s)"
shows "wp (sinit i) P E s"
⟨*proof*⟩

lemma *wp_sdecl[wprules]:*
assumes *"⋀x51 x52. t = SType.TArray x51 x52 ⟹ wp (sinit i) P E s"*
and *"⋀x6. t = SType.DArray x6 ⟹ wp (sinit i) P E s"*
and *"⋀x71 x72. t = SType.TMap x71 x72 ⟹ wp (sinit i) P E s"*
and *"⋀x8. t = SType.TEnum x8 ⟹ wp (sinit i) P E s"*
and *"⋀x. t = SType.TValue x ⟹ E Err s"*
shows *"wp (sdecl t i) P E s"*
⟨*proof*⟩

lemma **(in** *Contract)* *wp_initStorage[wprules]:*
assumes *"P Empty (storage_update i v s)"*
shows *"wp (initStorage i v) P E s"*
⟨*proof*⟩

lemma **(in** *Solidity)* *wp_init_balance[wprules]:*
assumes *"P Empty (balance_update (Balances s this + unat msg_value) s)"*
shows *"wp init_balance P E s"*
⟨*proof*⟩

lemma **(in** *Solidity)* *wp_init_balance_np[wprules]:*
assumes *"P Empty (balance_update (Balances s this) s)"*
shows *"wp init_balance_np P E s"*
⟨*proof*⟩

lemma **(in** *Solidity)* *wp_cinit[wprules]:*
assumes *"P Empty (calldata_update i c (stack_update i (kdata.Calldata (Some (|Location = i, Offset = []|))) s))"*
shows *"wp (cinit (c:: 'a valtype call_data) i) P E s"*
⟨*proof*⟩

lemma **(in** *Contract)* *wp_assign_stack_monad[wprules]:*
assumes *"wp m (λa. wp (lfold is ≫= (λis. assign_stack i is a ≫= (λ_. return Empty))) P E) E s"*
shows *"wp (assign_stack_monad i is m) P E s"*
⟨*proof*⟩

lemma **(in** *Contract)* *wp_storage_update_monad[wprules]:*
assumes *"⋀y. updateStore (xs @ is) sd (state.Storage s this p) = Some y ⟹ P Empty (storage_update p y s)"*
and *"updateStore (xs @ is) sd (state.Storage s this p) = None ⟹ E Err s"*
shows *"wp (storage_update_monad xs is sd p) P E s"*
⟨*proof*⟩

lemma **(in** *Contract)* *wp_assign_storage1[wperules]:*
assumes *"y = rvalue.Value v"*
and *"wp (storage_update_monad [] is (K (storage_data.Value v)) i) P E s"*
shows *"wp (assign_storage i is y) P E s"*
⟨*proof*⟩

lemma **(in** *Contract)* *wp_assign_storage2[wprules]:*
assumes *"wp (storage_update_monad [] is (K (storage_data.Value v)) i) P E s"*
shows *"wp (assign_storage i is (rvalue.Value v)) P E s"*
⟨*proof*⟩

lemma **(in** *Contract)* *wp_assign_storage_monad[wprules]:*
assumes *"wp m (λa. wp (lfold is ≫= (λis. assign_storage i is a)) P E) E s"*
shows *"wp (assign_storage_monad i is m) P E s"*
⟨*proof*⟩

lemma **(in** *Contract)* *wp_stackLookup[wprules]:*
assumes *"wp (lfold es)*
(λa. wp (stack_disjoint x (λk. return (rvalue.Value k))
(λp. option Err (λs. mlookup (state.Memory s) a p) ≫=
(λl. option Err (λs. state.Memory s $ l) ≫=

```
                        (λmd. if mdata.is_Value md then return (rvalue.Value (mdata.vt md))
                              else return (rvalue.Memory l))))
              (λp xs.
                  option Err (λs. state.Calldata s $$ p ≫= clookup (xs @ a)) ≫=
                  (λsd. if call_data.is_Value sd then return (rvalue.Value (call_data.vt sd))
                        else return (rvalue.Calldata (Some (|Location = p, Offset = xs @ a|)))))
              (return (rvalue.Calldata None))
              (λp xs.
                  option Err (λs. slookup (xs @ a) (state.Storage s this p)) ≫=
                  (λsd. if storage_data.is_Value sd then return (rvalue.Value (storage_data.vt sd))
                        else return (rvalue.Storage (Some (|Location = p, Offset = xs @ a|)))))
              (return (rvalue.Storage None)))
          P E)
    E s"
  shows "wp (stackLookup x es) P E s"
  ⟨proof⟩

lemma (in Keccak256) wp_keccak256[wprules]:
  assumes "wp m (λa. wp (return (keccak256 a)) P E) E s"
  shows "wp (keccak256_monad m) P E s"
  ⟨proof⟩

lemma (in External) wp_transfer_monad[wprules]:
  assumes " wp am
    (λa. wp (readValue a ≫=
             (λav. readAddress av ≫=
                   (λa. vm ≫=
                        (λvk. readValue vk ≫=
                              (λvv. readSint vv ≫=
                                    (λv.
assert Err (λs. unat v ≤ Balances s this) ≫=
(λ_. modify (λs. balance_update (Balances s this - unat v) s) ≫=
      (λ_. modify (λs. balances_update a (Balances s a + unat v) s) ≫=
            (λ_. ecall (external call)))))))))
            P E)
    E s"
  shows "wp (transfer_monad call am vm) P E s"
  ⟨proof⟩

lemma wp_readValue[wprules]:
  assumes "P (storage_data.vt yp) s"
  shows "wp (readValue (rvalue.Value (storage_data.vt yp))) P E s"
  ⟨proof⟩

lemma wp_readAddress[wprules]:
  assumes "P yp s"
  shows "wp (readAddress (Address yp)) P E s"
  ⟨proof⟩

lemma wp_stackCheck[wprules]:
  assumes "⋀p. Stack s $$ i = Some (kdata.Storage (Some p)) ⟹ wp (sf (Location p) (Offset p)) P E s"
      and "⋀l. Stack s $$ i = Some (kdata.Memory l) ⟹ wp (mf l) P E s"
      and "⋀p. Stack s $$ i = Some (kdata.Calldata (Some p)) ⟹ wp (cf (Location p) (Offset p)) P E
s"
      and "⋀v. Stack s $$ i = Some (kdata.Value v) ⟹ wp (kf v) P E s"
      and "Stack s $$ i = None ⟹ E Err s"
      and "Stack s $$ i = Some (kdata.Storage None) ⟹ wp sp P E s"
      and "Stack s $$ i = Some (kdata.Calldata None) ⟹ wp cp P E s"
    shows "wp (stack_disjoint i kf mf cf cp sf sp) P E s"
  ⟨proof⟩

lemma execute_normal:
  assumes "execute x s = Normal (a, b)"
  shows "effect x s (Inl (a,b))" ⟨proof⟩
```

lemma *execute_exception:*
assumes *"execute x s = Exception (a, b)"*
shows *"effect x s (Inr (a,b))"* ⟨*proof*⟩

lemma (**in** *Contract*) *inv_wp:*
assumes *"effect m s r"*
and *"wp m (K x) (K y) s"*
shows *"inv r x y"*
⟨*proof*⟩

lemma (**in** *Contract*) *post_wp:*
assumes *"effect m s r"*
and *"wp m (λr s'. P s r s' ∧ inv_state Is s') (K (inv_state Ie)) s"*
shows *"post s r Is Ie P"*
⟨*proof*⟩

lemma (**in** *Contract*) *wp_storeArrayLength[wprules]:*
assumes *"wp (lfold xs)*
(λa. wp (option Err (λs. slookup a (state.Storage s this v)) ≫=
(λsd. storage_disjoint sd (K (throw Err)) (λsa. return (rvalue.Value (Uint (word_of_nat
(length (storage_data.ar sd)))))) (K (throw Err))))
P E)
E s"
shows *"wp (storeArrayLength v xs) P E s"*
⟨*proof*⟩

lemma (**in** *Contract*) *wp_arrayLength[wprules]:*
assumes *"wp (lfold xs)*
(λa. wp (stack_disjoint v (K (throw Err))
(λp. option Err (λs. mlookup (state.Memory s) a p) ≫=
(λl. option Err (λs. state.Memory s $ l) ≫=
(λmd. if mdata.is_Array md
then return (rvalue.Value (Uint (word_of_nat (length (mdata.ar
md))))) else throw Err)))
(λp xs.
option Err (λs. state.Calldata s $$ p ≫= clookup (xs @ a)) ≫=
(λsd. if call_data.is_Array sd then return (rvalue.Value (Uint (word_of_nat (length
(call_data.ar sd)))))
else throw Err))
(throw Err)
(λp xs.
option Err (λs. slookup (xs @ a) (state.Storage s this p)) ≫=
(λsd. if storage_data.is_Array sd then return (rvalue.Value (Uint (word_of_nat
(length (storage_data.ar sd)))))
else throw Err))
(throw Err))
P E)
E s"
shows *"wp (arrayLength v xs) P E s"*
⟨*proof*⟩

lemma (**in** *Contract*) *wp_storearrayLength[wprules]:*
assumes *"slookup [] (state.Storage s this STR ''proposals'') = None ⟹ E Err s"*
and *"wp (storage_disjoint (state.Storage s this STR ''proposals'') (K (throw Err))*
(λsa. return (rvalue.Value (Uint (word_of_nat (length (storage_data.ar (state.Storage s this
STR ''proposals''))))))) (K (throw Err)))
P E s"
shows *"wp (storeArrayLength STR ''proposals'' []) P E s"*
⟨*proof*⟩

lemma (**in** *Contract*) *wp_storage_disjoint[wprules]:*
assumes *"⋀v. sd = storage_data.Value v ⟹ wp (vf v) P E s"*
and *"⋀a. sd = storage_data.Array a ⟹ wp (af a) P E s"*

```
    and "⋀m. sd = storage_data.Map m ⟹ wp (mf m) P E s"
  shows "wp (storage_disjoint sd vf af mf) P E s"
  ⟨proof⟩

lemma (in Contract) wp_allocate[wprules]:
  assumes "wp (lfold es)
     (λa. wp (option Err (λs. slookup a (state.Storage s this i) ≫= push d) ≫=
               (λar. storage_update_monad [] a (K ar) i))
           P E)
     E s"
  shows "wp (allocate i es d) P E s"
  ⟨proof⟩

lemma (in Contract) wp_create_memory_array[wprules]:
  assumes "wp sm
     (λa. wp (case a of
               rvalue.Value (Uint s') ⇒
                 Solidity.write (adata.Array (array (unat s') (cdefault t))) i
               | rvalue.Value _ ⇒ throw Err | _ ⇒ throw Err)
           P E)
     E s"
  shows "wp (create_memory_array i t sm) P E s"
  ⟨proof⟩
```

Using postconditions for WP

```
lemma (in Solidity) wp_post:
  assumes "(⋀r. effect (c x) s r ⟹ post s r (K True) (K True) P')"
      and "⋀a sa. P' s a sa ⟹ P a sa"
      and "⋀sa e. Q e sa"
    shows "wp (c x) P Q s"
  ⟨proof⟩

declare(in Contract) wp_stackCheck[wprules del]

lemma (in Contract) wp_assign_stack_kdvalue[wprules]:
  assumes "Stack s $$ i = None ⟹ E Err s"
      and "¬(∃x2. Stack s $$ i = Some (kdata.Memory x2))"
      and "Stack s $$ i = Some (kdata.Storage None) ⟹ E Err s"
      and "Stack s $$ i = Some (kdata.Calldata None) ⟹ E Err s"
      and "⋀aa. Stack s $$ i = Some (kdata.Storage (Some aa)) ⟹
          wp (storage_update_monad (Offset aa) is (K (storage_data.Value v)) (Location aa)) P E s"
      and "⋀x4. Stack s $$ i = Some (kdata.Value x4) ⟹
          wp (modify (stack_update i (kdata.Value v)) ≫= (λa. return Empty)) P E s"
      and "⋀a. Stack s $$ i = Some (kdata.Calldata (Some a)) ⟹ E Err s"
    shows "wp (assign_stack i is (rvalue.Value v)) P E s"
  ⟨proof⟩
declare(in Contract) wp_stackCheck[wprules]

declare write.simps [simp del]
declare mupdate.simps [simp del]
declare mlookup.simps [simp del]
declare alookup.simps [simp del]
declare locations.simps [simp del]

end
```

7 Unit Tests

In this chapter, we validate the compliance of the semantics with the official Solidity specification. To this end we implemented a set of unit tests which were also executed on the Remix IDE.

theory *Unit_Tests*
imports *Solidity "HOL-Library.Code_Target_Numeral" "HOL-Library.Code_Target_Nat" "HOL-Library.Code_Target_Int"*
begin

7.1 Examples (Unit_Tests)

definition *"vt_true = Bool True"*
definition *"vt_false = Bool False"*
definition *"vt_sint_m10 = Uint (-10)"*
definition *"vt_sint_0 = Uint 0"*
definition *"vt_sint_1 = Uint 1"*
definition *"vt_sint_2 = Uint 2"*
definition *"vt_sint_10 = Uint 10"*
definition *"vt_address = Address null"*

definition *"sd_true = storage_data.Value vt_true"*
definition *"sd_false = storage_data.Value vt_false"*
definition *"sd_sint8_m10 = storage_data.Value vt_sint_m10"*
definition *"sd_uint8_10 = storage_data.Value vt_sint_10"*
definition *"sd_address = storage_data.Value vt_address"*
definition *"(sd_Array_3_true::aspace valtype storage_data) = storage_data.Array [sd_true,sd_true,sd_true]"*
definition *"(sd_Array_3_false::aspace valtype storage_data) = storage_data.Array [sd_false,sd_false,sd_false]"*
definition *"(sd_Array_2_3_true_false::aspace valtype storage_data) = storage_data.Array [sd_Array_3_true, sd_Array_3_false]"*
definition *"(sd_Array_2_3_false_false::aspace valtype storage_data) = storage_data.Array [sd_Array_3_false, sd_Array_3_false]"*

definition *"md_true = mdata.Value vt_true"*
definition *"md_false = mdata.Value vt_false"*
definition *"md_sint_m10 = mdata.Value vt_sint_m10"*
definition *"md_uint_10 = mdata.Value vt_sint_10"*
definition *"md_address = mdata.Value vt_address"*
definition *"mem_Array_2_3_true_false = [md_true,md_true,md_true,mdata.Array [0,1,2],md_false,md_false,md_false,mdata [4,5,6],mdata.Array [3,7]]"*
definition *"mem_Array_2_3_false_false = [md_false,md_false,md_false,mdata.Array [0,1,2],md_false,md_false,md_false,mdata.Array [4,5,6],mdata.Array [3,7]]"*
definition *"mem_sint_m10_uint_10= [md_sint_m10,md_uint_10]"*

definition *"cd_true = call_data.Value vt_true"*
definition *"cd_false = call_data.Value vt_false"*
definition *"cd_sint8_m10 = call_data.Value vt_sint_m10"*
definition *"cd_uint8_10 = call_data.Value vt_sint_10"*
definition *"cd_address = call_data.Value vt_address"*
definition *"cd_Array_3_true = call_data.Array [cd_true,cd_true,cd_true]"*
definition *"cd_Array_3_false = call_data.Array [cd_false,cd_false,cd_false]"*
definition *"cd_Array_2_3_true_false = call_data.Array [cd_Array_3_true, cd_Array_3_false]"*
definition *"cd_Array_2_3_false_false = call_data.Array [cd_Array_3_false, cd_Array_3_false]"*

global_interpretation *method: Method A1 250 100*

```
  defines method_sender_monad = method.sender_monad
      and method_value_monad  = method.value_monad
      and method_timestamp_monad  = method.block_timestamp_monad
  ⟨proof⟩

global_interpretation contract: Contract A1
  defines contract_assign_stack_monad = contract.assign_stack_monad
      and contract_assign_stack       = contract.assign_stack
      and contract_stackLookup        = contract.stackLookup
      and contract_storeLookup        = contract.storeLookup
      and contract_assign_storage_monad = contract.assign_storage_monad
      and contract_assign_storage = contract.assign_storage
      and contract_storage_update_monad = contract.storage_update_monad
      and contract_storage_update = contract.storage_update
      and contract_this_monad = contract.this_monad
      and contract_storearrayLength = contract.storeArrayLength
      and contract_arrayLength = contract.arrayLength
      and contract_allocate = contract.allocate
  ⟨proof⟩

global_interpretation keccak256: Keccak256 id
  defines keccak256_keccak256_monad = keccak256.keccak256_monad
  ⟨proof⟩
```

7.2 States (Unit_Tests)

```
definition emptyState::"aspace state" where
"emptyState =
  (|
    state.Memory = [],
    state.Calldata = fmempty,
    state.Storage = (λ_ _. undefined),
    state.Stack = fmempty,
    state.Balances = (λ_. 0)
  |)"

definition m where
  "m = [md_true,
        md_true,
        md_true,
        mdata.Array [0,1,2],
        md_false,
        md_false,
        md_false,
        mdata.Array [4,5,6],
        mdata.Array [3,7],
        md_true,
        md_true,
        md_true,
       mdata.Array [9,10,11]]"

definition m' where
  "m' = [md_true,
        md_true,
        md_true,
        mdata.Array [9,10,11],
        md_false,
        md_false,
        md_false,
        mdata.Array [4,5,6],
        mdata.Array [3,7],
        md_true,
        md_true,
        md_true,
```

```
      mdata.Array [9,10,11]]"
definition "mystack = fmap_of_list [(STR ''x'', kdata.Memory 8), (STR ''y'', kdata.Memory 12)]"
definition "mystate = emptyState(|Stack := mystack, Memory := m|)"
```

7.3 Constants (Unit_Tests)

```
lemma "P (execute true_monad emptyState)
     = P (Normal (rvalue.Value (Bool True),emptyState))"
```
⟨*proof*⟩

```
lemma "P (execute false_monad emptyState)
     = P (Normal (rvalue.Value (Bool False),emptyState))"
```
⟨*proof*⟩

```
lemma "P (execute (sint_monad 5) emptyState)
     = P (Normal (rvalue.Value (Uint 5),emptyState))"
```
⟨*proof*⟩

```
lemma "P (execute (address_monad A5) emptyState)
     = P (Normal (rvalue.Value (Address A5),emptyState))"
```
⟨*proof*⟩

```
lemma "is_Normal (execute (do {
        assert_monad (equals_monad (contract_this_monad) (address_monad A1))
      }) emptyState)"
```
⟨*proof*⟩

```
lemma "is_Normal (execute (do {
        assert_monad (equals_monad (method_sender_monad) (address_monad A1))
      }) emptyState)"
```
⟨*proof*⟩

```
lemma "is_Normal (execute (do {
        assert_monad (equals_monad (method_value_monad) (sint_monad 250))
      }) emptyState)"
```
⟨*proof*⟩

```
lemma "is_Normal (execute (do {
        assert_monad (equals_monad (method_timestamp_monad) (sint_monad 100))
      }) emptyState)"
```
⟨*proof*⟩

```
lemma "is_Normal (execute (do {
        assert_monad (equals_monad (keccak256_keccak256_monad (sint_monad 5)) (sint_monad 5))
      }) emptyState)"
```
⟨*proof*⟩

7.4 Basic Operators (Unit_Tests)

7.4.1 Not monad

```
lemma "is_Normal (execute (do {
        assert_monad (equals_monad (not_monad true_monad) false_monad)
      }) emptyState)"
```
⟨*proof*⟩

```
lemma "is_Normal (execute (do {
        assert_monad (equals_monad (not_monad false_monad) true_monad)
      }) emptyState)"
```
⟨*proof*⟩

7.4.2 Equals monad

```
lemma "is_Normal (execute (do {
        assert_monad (equals_monad (sint_monad 10) (sint_monad 10))
```

```
    }) emptyState)"
```
⟨*proof*⟩

7.4.3 Less monad

lemma *"is_Normal (execute (do {*
assert_monad (less_monad (sint_monad 10) (sint_monad 11))
}) emptyState)"
⟨*proof*⟩

7.4.4 And monad

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (and_monad (true_monad) (false_monad)) (false_monad))
}) emptyState)"
⟨*proof*⟩

7.4.5 Or monad

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (or_monad (true_monad) (false_monad)) (true_monad))
}) emptyState)"
⟨*proof*⟩

7.4.6 Plus monad

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (plus_monad (sint_monad 5) (sint_monad 6)) (sint_monad 11))
}) emptyState)"
⟨*proof*⟩

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (plus_monad (sint_monad (2^256 - 1)) (sint_monad 6)) (sint_monad 5))
}) emptyState)"
⟨*proof*⟩

7.4.7 Plus monad safe

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (plus_monad_safe (sint_monad 5) (sint_monad 6)) (sint_monad 11))
}) emptyState)"
⟨*proof*⟩

lemma *"is_Exception (execute (do {*
assert_monad (equals_monad (plus_monad_safe (sint_monad (2^256 - 1)) (sint_monad 6)) (sint_monad
5))
}) emptyState)"
⟨*proof*⟩

7.4.8 Minus monad

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (minus_monad (sint_monad 6) (sint_monad 5)) (sint_monad 1))
}) emptyState)"
⟨*proof*⟩

7.4.9 Minus monad safe

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (minus_monad_safe (sint_monad 6) (sint_monad 5)) (sint_monad 1))
}) emptyState)"
⟨*proof*⟩

lemma *"is_Exception (execute (do {*

```
        assert_monad (equals_monad (minus_monad_safe (sint_monad 5) (sint_monad 6)) (sint_monad (2^256 -
5)))
      }) emptyState)"
```
⟨proof⟩

7.4.10 Mult monad

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (mult_monad (sint_monad 5) (sint_monad 6)) (sint_monad 30))
}) emptyState)"
⟨proof⟩

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (mult_monad (sint_monad (2^255)) (sint_monad 2)) (sint_monad 0))
}) emptyState)"
⟨proof⟩

7.4.11 Mult monad safe

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (mult_monad_safe (sint_monad 5) (sint_monad 6)) (sint_monad 30))
}) emptyState)"
⟨proof⟩

lemma *"is_Exception (execute (do {*
assert_monad (equals_monad (mult_monad_safe (sint_monad (2^255)) (sint_monad 2)) (sint_monad 0))
}) emptyState)"
⟨proof⟩

7.4.12 Mod monad

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (mod_monad (sint_monad 9) (sint_monad 5)) (sint_monad 4))
}) emptyState)"
⟨proof⟩

7.5 Store lookup (Unit_Tests)

7.5.1 Value type

definition *"pstorage1 = (λ_. undefined) (STR ''x'' := storage_data.Value (Uint 5))"*
definition *"storage1 = (λ_. undefined) (A1 := pstorage1)"*
definition *"state1 = emptyState(|Storage := storage1|)"*

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (contract_storeLookup (STR ''x'') []) (sint_monad 5))
}) state1)" ⟨proof⟩

7.5.2 Array

definition *"pstorage2 = (λ_. undefined) (STR ''x'' := storage_data.Array [storage_data.Value (Uint 5)])"*
definition *"storage2 = (λ_. undefined) (A1 := pstorage2)"*
definition *"state2 = emptyState(|Storage := storage2|)"*

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (contract_storeLookup (STR ''x'') [sint_monad 0]) (sint_monad 5))
}) state2)" ⟨proof⟩

definition *"pstorage20 = (λ_. undefined) (STR ''x'' := storage_data.Array [storage_data.Array*
[storage_data.Value (Uint 5)]])"
definition *"storage20 = (λ_. undefined) (A1 := pstorage20)"*
definition *"state20 = emptyState(|Storage := storage20|)"*

lemma *"is_Normal (execute (do {
assert_monad (equals_monad (contract_storeLookup (STR ''x'') [sint_monad 0, sint_monad 0]) (sint_monad 5))
}) state20)"* ⟨*proof*⟩

definition *"pstorage21 = pstorage20 (STR ''y'' := storage_data.Array [storage_data.Array [storage_data.Value (Uint 5)]])"*
definition *"storage21 = (λ_. undefined) (A1 := pstorage21)"*
definition *"state21 = emptyState(|Storage := storage21|)"*

lemma *"is_Normal (execute (do {
assert_monad (equals_monad (contract_storeLookup (STR ''x'') [sint_monad 0, sint_monad 0]) (contract_storeLookup (STR ''y'') [sint_monad 0, sint_monad 0]))
}) state21)"* ⟨*proof*⟩

lemma *"is_Exception (execute (do {
assert_monad (equals_monad (contract_storeLookup (STR ''x'') [sint_monad 0]) (contract_storeLookup (STR ''y'') [sint_monad 0]))
}) state21)"* ⟨*proof*⟩

Mappings

definition *"pstorage3 = (λ_. undefined) (STR ''x'' := storage_data.Map ((λ_. undefined) (Uint (0) := storage_data.Value (Uint 5))))"*
definition *"storage3 = (λ_. undefined) (A1 := pstorage3)"*
definition *"state3 = emptyState(|Storage := storage3|)"*

lemma *"is_Normal (execute (do {
assert_monad (equals_monad (contract_storeLookup (STR ''x'') [sint_monad 0]) (sint_monad 5))
}) state3)"* ⟨*proof*⟩

definition *"pstorage30 = (λ_. undefined) (STR ''x'' := storage_data.Map ((λ_. undefined) (Uint 0 := storage_data.Map ((λ_. undefined) (Bool False := (storage_data.Value (Uint 5)))))))"*
definition *"storage30 = (λ_. undefined) (A1 := pstorage30)"*
definition *"state30 = emptyState(|Storage := storage30|)"*

lemma *"is_Normal (execute (do {
assert_monad (equals_monad (contract_storeLookup (STR ''x'') [sint_monad 0, false_monad]) (sint_monad 5))
}) state30)"* ⟨*proof*⟩

7.6 Stack lookup (Unit_Tests)

7.6.1 Value type

lemma *"is_Normal (execute (do {
init (Uint 5) (STR ''x'');
assert_monad (equals_monad (contract_stackLookup (STR ''x'') []) (sint_monad 5))
}) emptyState)"* ⟨*proof*⟩

7.6.2 Memory

lemma *"is_Normal (execute (do {
write (adata.Array [adata.Value (Uint 5)]) (STR ''x'');
assert_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0]) (sint_monad 5))
}) emptyState)"* ⟨*proof*⟩

lemma *"is_Normal (execute (do {*
write (adata.Array [adata.Array [adata.Value (Uint 5)]]) (STR ''x'');
write (adata.Array [adata.Value (Uint 0)]) (STR ''y'');
contract_assign_stack_monad (STR ''y'') [] (contract_stackLookup (STR ''x'') [sint_monad 0]);
assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0]) (sint_monad 5))
}) emptyState)" ⟨*proof*⟩

lemma *"is_Normal (execute (do {*
write (adata.Array [adata.Array [adata.Value (Uint 5)]]) (STR ''x'');
write (adata.Array [adata.Value (Uint 0)]) (STR ''y'');
contract_assign_stack_monad (STR ''y'') [sint_monad 0] (contract_stackLookup (STR ''x'')
[sint_monad 0]);
assert_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0, sint_monad 0])
(sint_monad 5));
assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0, sint_monad 0])
(sint_monad 5))
}) emptyState)" ⟨*proof*⟩

7.6.3 Calldata

lemma *"is_Normal (execute (do {*
cinit (call_data.Array [call_data.Value (Uint 5)]) (STR ''x'');
assert_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0]) (sint_monad 5))
}) emptyState)" ⟨*proof*⟩

definition *"pcalldata71 = fmap_of_list [(STR ''x'', call_data.Array [call_data.Array [call_data.Value*
(Uint 5)]])]"
definition *"pstack71 = fmap_of_list [(STR ''y'', kdata.Calldata (Some (|Location = STR ''x'', Offset =*
[Uint 0]|)))]"
definition *"state71 = emptyState(|Calldata := pcalldata71, Stack:=pstack71|)"*
lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0]) (sint_monad 5))
}) state71)" ⟨*proof*⟩

Storage pointers

definition *"pstorage8 = (λ_. undefined) (STR ''x'' := storage_data.Array [storage_data.Value (Uint 5)])"*
definition *"storage8 = (λ_. undefined) (A1 := pstorage8)"*
definition *"stack8 = fmap_of_list [(STR ''y'', kdata.Storage (Some (|Location = STR ''x'', Offset=*
[]|)))]"
definition *"state8 = emptyState(|Storage := storage8, Stack := stack8|)"*

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0]) (sint_monad 5))
}) state8)" ⟨*proof*⟩

definition *"pstorage9 = (λ_. undefined) (STR ''x'' := storage_data.Array [storage_data.Array*
[storage_data.Value (Uint 5)]])"
definition *"storage9 = (λ_. undefined) (A1 := pstorage9)"*
definition *"stack9 = fmap_of_list [(STR ''y'', kdata.Storage (Some (|Location = STR ''x'', Offset= [Uint*
0]|)))]"
definition *"state9 = emptyState(|Storage := storage9, Stack := stack9|)"*

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0]) (sint_monad 5))

```
}) state9)" ⟨proof⟩
```

7.6.4 Arrays

Storage Arrays

Length **definition** *"pstorage10 = (λ_. undefined) (STR ''x'' := storage_data.Array [storage_data.Value (Uint 5)])"*
definition *"storage10 = (λ_. undefined) (A1 := pstorage10)"*
definition *"state10 = emptyState⦇Storage := storage10⦈"*

Push **lemma** *"is_Normal (execute (do {*
assert_monad (equals_monad (contract_storearrayLength (STR ''x'') []) (sint_monad 1))
}) state10)" ⟨proof⟩

definition *"pstorage11 = (λ_. undefined) (STR ''x'' := storage_data.Array [storage_data.Value (Uint 5)])"*
definition *"storage11 = (λ_. undefined) (A1 := pstorage11)"*
definition *"state11 = emptyState⦇Storage := storage11⦈"*

lemma *"is_Normal (execute (do {*
contract_allocate (STR ''x'') [] (storage_data.Value (Uint 6));
assert_monad (equals_monad (contract_storeLookup (STR ''x'') [sint_monad 0]) (sint_monad 5));
assert_monad (equals_monad (contract_storeLookup (STR ''x'') [sint_monad 1]) (sint_monad 6))
}) state11)" ⟨proof⟩

Memory Arrays

lemma *"is_Normal (execute (do {*
write (adata.Array [adata.Value (Uint 1)]) (STR ''x'');
assert_monad (equals_monad (contract_arrayLength (STR ''x'') []) (sint_monad 1));
assert_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0]) (sint_monad 1))
}) emptyState)" ⟨proof⟩

lemma *"is_Normal (execute (do {*
write (adata.Array [adata.Array [adata.Value (Uint 1)]]) (STR ''x'');
assert_monad (equals_monad (contract_arrayLength (STR ''x'') []) (sint_monad 1));
assert_monad (equals_monad (contract_arrayLength (STR ''x'') [sint_monad 0]) (sint_monad 1));
assert_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0, sint_monad 0]) (sint_monad 1))
}) emptyState)" ⟨proof⟩

Calldata Arrays

lemma *"is_Normal (execute (do {*
cinit (call_data.Array [call_data.Array [call_data.Value (Uint 5)]]) (STR ''x'');
assert_monad (equals_monad (contract_arrayLength (STR ''x'') []) (sint_monad 1))
}) emptyState)" ⟨proof⟩

Storage pointer Arrays

definition *"pstorage14 = (λ_. undefined) (STR ''x'' := storage_data.Array [storage_data.Value (Uint 5)])"*
definition *"storage14 = (λ_. undefined) (A1 := pstorage14)"*
definition *"stack14 = fmap_of_list [(STR ''y'', kdata.Storage (Some ⦇Location = STR ''x'', Offset= []⦈))]"*
definition *"state14 = emptyState⦇Storage := storage14, Stack := stack14⦈"*

lemma *"is_Normal (execute (do {*
assert_monad (equals_monad (contract_arrayLength (STR ''y'') []) (sint_monad 1))
}) state14)" ⟨proof⟩

7.7 Conditionals (Unit_Tests)

lemma *"is_Normal (execute (do {*
init (Uint 0) (STR ''x'');
cond_monad (true_monad)
(contract_assign_stack_monad (STR ''x'') [] (sint_monad 1))
(contract_assign_stack_monad (STR ''x'') [] (sint_monad 2));
assert_monad (equals_monad (contract_stackLookup (STR ''x'') []) (sint_monad 1))
}) emptyState)"
⟨*proof*⟩

lemma *"is_Normal (execute (do {*
init (Uint 0) (STR ''x'');
cond_monad (false_monad)
(contract_assign_stack_monad (STR ''x'') [] (sint_monad 1))
(contract_assign_stack_monad (STR ''x'') [] (sint_monad 2));
assert_monad (equals_monad (contract_stackLookup (STR ''x'') []) (sint_monad 2))
}) emptyState)"
⟨*proof*⟩

7.8 Assignments (Unit_Tests)

7.8.1 Stack Value to Stack Value

lemma *"is_Normal (execute (do {*
init (Uint 0) (STR ''x'');
init (Uint 0) (STR ''y'');
contract_assign_stack_monad (STR ''y'') [] (sint_monad 1);
assert_monad (equals_monad (contract_stackLookup (STR ''x'') []) (sint_monad 0));
assert_monad (equals_monad (contract_stackLookup (STR ''y'') []) (sint_monad 1))
}) emptyState)"
⟨*proof*⟩

7.8.2 Memory Array to Memory Array

lemma *"is_Normal (execute (do {*
write (adata.Array [adata.Value (Uint 0)]) (STR ''x'');
write (adata.Array [adata.Value (Uint 0)]) (STR ''y'');
contract_assign_stack_monad (STR ''y'') [] (contract_stackLookup (STR ''x'') []);
contract_assign_stack_monad (STR ''x'') [sint_monad 0] (sint_monad 1);
assert_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0]) (sint_monad 1));
assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0]) (sint_monad 1))
}) emptyState)"
⟨*proof*⟩

7.8.3 Calldata Array to Memory Array

lemma *"is_Normal (execute (do {*
cinit (call_data.Array [call_data.Value (Uint 0)]) (STR ''x'');
write (adata.Array [adata.Value (Uint 0)]) (STR ''y'');
require_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0]) (sint_monad 0));
contract_assign_stack_monad (STR ''y'') [] (contract_stackLookup (STR ''x'') []);
contract_assign_stack_monad (STR ''y'') [sint_monad 0] (sint_monad 1);
assert_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0]) (sint_monad 0));
assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0]) (sint_monad 1))
}) emptyState)"
⟨*proof*⟩

lemma *"is_Normal (execute (do {*
cinit (call_data.Array [call_data.Array [call_data.Value (Uint 0)]]) (STR ''x'');
write (adata.Array [adata.Array [adata.Value (Uint 1)]]) (STR ''y'');
write (adata.Array [adata.Value (Uint 0)]) (STR ''z'');
contract_assign_stack_monad (STR ''z'') [] (contract_stackLookup (STR ''y'') [sint_monad 0]);

```
        contract_assign_stack_monad (STR ''y'') [] (contract_stackLookup (STR ''x'') []);
        assert_monad (equals_monad (contract_stackLookup (STR ''z'') [sint_monad 0]) (sint_monad 1));
        assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0, sint_monad 0])
(sint_monad 0))
      }) emptyState)"
```
⟨*proof*⟩

```
lemma "is_Normal (execute (do {
        cinit (call_data.Array [call_data.Array [call_data.Array [call_data.Value (Uint 0)]]]) (STR
''x'');
        write (adata.Array [adata.Array [adata.Array [adata.Value (Uint 1)]]]) (STR ''y'');
        write (adata.Array [adata.Array [adata.Value (Uint 0)]]) (STR ''z'');
        contract_assign_stack_monad (STR ''z'') [sint_monad 0] (contract_stackLookup (STR ''y'')
[sint_monad 0, sint_monad 0]);
        contract_assign_stack_monad (STR ''y'') [sint_monad 0] (contract_stackLookup (STR ''x'')
[sint_monad 0]);
        assert_monad (equals_monad (contract_stackLookup (STR ''z'') [sint_monad 0, sint_monad 0])
(sint_monad 1));
        assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0, sint_monad 0,
sint_monad 0]) (sint_monad 0))
      }) emptyState)"
```
⟨*proof*⟩

7.8.4 Storage Pointer Array to Memory Array

```
definition "pstorage17 = (λ_. undefined) (STR ''s'' := storage_data.Array [storage_data.Value (Uint
0)])"
definition "storage17 = (λ_. undefined) (A1 := pstorage17)"
definition "stack17 = fmap_of_list [(STR ''x'', kdata.Storage (Some (|Location = STR ''s'', Offset=
[]|)))]"
definition "state17 = emptyState(|Storage := storage17, Stack := stack17|)"

lemma "is_Normal (execute (do {
        require_monad (equals_monad (contract_storeLookup (STR ''s'') [sint_monad 0]) (sint_monad 0));
        write (adata.Array [adata.Value (Uint 0)]) (STR ''y'');
        contract_assign_stack_monad (STR ''y'') [] (contract_stackLookup (STR ''x'') []);
        contract_assign_stack_monad (STR ''y'') [sint_monad 0] (sint_monad 1);
        assert_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0]) (sint_monad 0));
        assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0]) (sint_monad 1))
      }) state17)" ⟨proof⟩
```

7.8.5 Storage Array to Memory Array

```
definition "pstorage18 = (λ_. undefined) (STR ''x'' := storage_data.Array [storage_data.Value (Uint
0)])"
definition "storage18 = (λ_. undefined) (A1 := pstorage18)"
definition "state18 = emptyState(|Storage := storage18|)"

lemma "is_Normal (execute (do {
        write (adata.Array [adata.Value (Uint 0)]) (STR ''y'');
        contract_assign_stack_monad (STR ''y'') [] (contract_storeLookup (STR ''x'') []);
        contract_assign_stack_monad (STR ''y'') [sint_monad 0] (sint_monad 1);
        assert_monad (equals_monad (contract_storeLookup (STR ''x'') [sint_monad 0]) (sint_monad 0));
        assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0]) (sint_monad 1))
      }) state18)" ⟨proof⟩
```

```
definition "pstorage181 = (λ_. undefined) (STR ''x'' := storage_data.Array [storage_data.Value (Uint
1)])"
definition "storage181 = (λ_. undefined) (A1 := pstorage181)"
definition "state181 = emptyState(|Storage := storage181|)"
```

```
lemma "is_Normal (execute (do {
        write (adata.Array [adata.Array [adata.Value (Uint 0)]]) (STR ''y'');
        contract_assign_stack_monad (STR ''y'') [sint_monad 0] (contract_storeLookup (STR ''x'') []);
        assert_monad (equals_monad (contract_storeLookup (STR ''x'') [sint_monad 0]) (sint_monad 1));
        assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0, sint_monad 0])
(sint_monad 1))
      }) state181)" ⟨proof⟩
```

7.8.6 Storage Pointer Array to Storage Pointer Array

```
definition "pstorage19 = (λ_. undefined) (STR ''s1'' := storage_data.Array [storage_data.Value (Uint
0)], STR ''s2'' := storage_data.Array [storage_data.Value (Uint 0)])"
definition "storage19 = (λ_. undefined) (A1 := pstorage19)"
definition "stack19 = fmap_of_list [
  (STR ''x'', kdata.Storage (Some (|Location = STR ''s1'', Offset= []|))),
  (STR ''y'', kdata.Storage (Some (|Location = STR ''s2'', Offset= []|)))
]"
definition "state19 = emptyState(|Storage := storage19, Stack := stack19|)"

lemma "is_Normal (execute (do {
        contract_assign_stack_monad (STR ''x'') [] (contract_stackLookup (STR ''y'') []);
        contract_assign_stack_monad (STR ''y'') [sint_monad 0] (sint_monad 1);
        assert_monad (equals_monad (contract_storeLookup (STR ''s1'') [sint_monad 0]) (sint_monad 0));
        assert_monad (equals_monad (contract_storeLookup (STR ''s2'') [sint_monad 0]) (sint_monad 1));
        assert_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0]) (sint_monad 1));
        assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0]) (sint_monad 1))
      }) state19)" ⟨proof⟩
```

7.8.7 Storage Array to Storage Pointer Array

```
definition "pstorage200 = (λ_. undefined) (STR ''s'' := storage_data.Array [storage_data.Value (Uint
0)])"
definition "storage200 = (λ_. undefined) (A1 := pstorage200)"
definition "stack200 = fmap_of_list [(STR ''x'', kdata.Storage None)]"
definition "state200 = emptyState(|Storage := storage200, Stack := stack200|)"

lemma "is_Normal (execute (do {
        contract_assign_stack_monad (STR ''x'') [] (contract_storeLookup (STR ''s'') []);
        contract_assign_storage_monad (STR ''s'') [sint_monad 0] (sint_monad 1);
        assert_monad (equals_monad (contract_storeLookup (STR ''s'') [sint_monad 0]) (sint_monad 1));
        assert_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0]) (sint_monad 1))
      }) state200)" ⟨proof⟩
```

7.8.8 Memory Array to Storage Array

```
lemma "is_Normal (execute ( do {
        write (adata.Array [adata.Value (Uint 0)]) (STR ''y'');
        contract_assign_storage_monad (STR ''x'') [] (contract_stackLookup (STR ''y'') []);
        contract_assign_storage_monad (STR ''x'') [sint_monad 0] (sint_monad 1);
        assert_monad (equals_monad (contract_storeLookup (STR ''x'') [sint_monad 0]) (sint_monad 1));
        assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0]) (sint_monad 0))
      }) state200)" ⟨proof⟩



lemma "is_Normal (execute ( do {
        contract_assign_stack_monad (STR ''x'') [] (contract_storeLookup (STR ''s'') []);
        write (adata.Array [adata.Value (Uint 1)]) (STR ''y'');
        contract_assign_storage_monad (STR ''s'') [] (contract_stackLookup (STR ''y'') []);
        assert_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0]) (sint_monad 1));
        assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0]) (sint_monad 1))
      }) state200)" ⟨proof⟩
```

7.8.9 Calldata Array to Storage Array

definition *"pstorage22 = (λ_. undefined) (STR ''s'' := (storage_data.Array [storage_data.Value (Uint 0)]))"*
definition *"storage22 = (λ_. undefined) (A1 := pstorage22)"*
definition *"state22 = emptyState⦇Storage := storage22⦈"*

lemma *"is_Normal (execute (do {*
cinit (call_data.Array [call_data.Value (Uint 0)]) (STR ''x'');
require_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0]) (sint_monad 0));
contract_assign_storage_monad (STR ''s'') [] (contract_stackLookup (STR ''x'') []);
contract_assign_storage_monad (STR ''s'') [sint_monad 0] (sint_monad 1);
assert_monad (equals_monad (contract_storeLookup (STR ''s'') [sint_monad 0]) (sint_monad 1));
assert_monad (equals_monad (contract_stackLookup (STR ''x'') [sint_monad 0]) (sint_monad 0))
}) state22)"
⟨*proof*⟩

7.8.10 Storage Array to Storage Array

definition *"pstorage23 = (λ_. undefined) (STR ''x'' := (storage_data.Array [storage_data.Value (Uint 0)]), STR ''y'' := (storage_data.Array [storage_data.Value (Uint 0)]))"*
definition *"storage23 = (λ_. undefined) (A1 := pstorage23)"*
definition *"state23 = emptyState⦇Storage := storage23⦈"*

lemma *"is_Normal (execute (do {*
contract_assign_storage_monad (STR ''x'') [] (contract_storeLookup (STR ''y'') []);
contract_assign_storage_monad (STR ''x'') [sint_monad 0] (sint_monad 1);
assert_monad (equals_monad (contract_storeLookup (STR ''x'') [sint_monad 0]) (sint_monad 1));
assert_monad (equals_monad (contract_storeLookup (STR ''y'') [sint_monad 0]) (sint_monad 0))
}) state23)" ⟨*proof*⟩

7.8.11 Storage Array to Memory Array

definition *"pstorage24 = (λ_. undefined) (STR ''x'' := storage_data.Array [storage_data.Value (Uint 1)])"*
definition *"storage24 = (λ_. undefined) (A1 := pstorage24)"*
definition *"state24 = emptyState⦇Storage := storage24⦈"*

lemma *"is_Normal (execute (do {*
write (adata.Array [adata.Array [adata.Value (Uint 0)]]) (STR ''a1'');
write (adata.Array [adata.Array [adata.Value (Uint 0)]]) (STR ''a2'');
contract_assign_stack_monad (STR ''a2'') [sint_monad 0] (contract_stackLookup (STR ''a1'') [sint_monad 0]);
contract_assign_stack_monad (STR ''a1'') [sint_monad 0] (contract_storeLookup (STR ''x'') []);
assert_monad (equals_monad (contract_stackLookup (STR ''a1'') [sint_monad 0, sint_monad 0]) (sint_monad 1));
assert_monad (equals_monad (contract_stackLookup (STR ''a2'') [sint_monad 0, sint_monad 0]) (sint_monad 0))
}) state24)" ⟨*proof*⟩

7.9 Declarations (Unit_Tests)

lemma *"is_Normal (execute (do {*
decl TSint (STR ''x'');
assert_monad (equals_monad (contract_stackLookup (STR ''x'') []) (sint_monad 0))
}) emptyState)" ⟨*proof*⟩

lemma *"is_Normal (execute (do {*
mdecl (TArray 1 (TValue TSint)) (STR ''y'');
assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0]) (sint_monad 0))
}) emptyState)" ⟨*proof*⟩

lemma *"state.Memory (snd (normal (execute (do {*
mdecl (DArray (TValue TSint)) (STR ''y'')
}) emptyState)))=[mdata.Array []]" ⟨*proof*⟩

lemma *"is_Exception (execute (do {*
sdecl (SType.TArray 1 (SType.TValue TSint)) (STR ''y'');
assert_monad (equals_monad (contract_stackLookup (STR ''y'') [sint_monad 0]) (sint_monad 0))
}) emptyState)" ⟨*proof*⟩

lemma
shows *"let pstorage =*
((λ_. undefined)
(STR ''x'' := storage_data.Value (Bytes [CHR 0x01, CHR 0x02, CHR 0x03])));
storage = (λ_. undefined) (A1 := pstorage);
state = emptyState(|Storage := storage|)
in is_Exception (execute (do {
assert_monad (equals_monad (bytes_index_monad (contract_storeLookup (STR ''x'') [])
(sint_monad 3)) (sint_monad 0))
}) state)"
⟨*proof*⟩

lemma
shows *"let pstorage =*
((λ_. undefined)
(STR ''x'' := storage_data.Value (Bytes [CHR 0xAA, CHR 0xBB, CHR 0xCC]))
(STR ''y'' := storage_data.Value (Bytes [CHR 0xBB]));
storage = (λ_. undefined) (A1 := pstorage);
state = emptyState(|Storage := storage|)
in is_Normal (execute (do {
assert_monad (equals_monad (bytes_index_monad (contract_storeLookup (STR ''x'') [])
(sint_monad 1)) (contract_storeLookup (STR ''y'') []))
}) state)"
⟨*proof*⟩

lemma
shows *"let pstorage =*
((λ_. undefined)
(STR ''x'' := storage_data.Value (Bytes [CHR 0xAA, CHR 0xBB, CHR 0xCC]))
(STR ''y'' := storage_data.Value (Bytes [CHR 0xBB]));
storage = (λ_. undefined) (A1 := pstorage);

```
           state = emptyState(|Storage := storage|)
        in is_Normal (execute (do {
             assert_monad (equals_monad (bytes_index_monad (contract_storeLookup (STR ''x'') [])
(sint_monad 1)) (contract_storeLookup (STR ''y'') []))
           }) state)"
```
⟨*proof*⟩

lemma
shows *"let pstorage =*
```
               ((λ_. undefined)
                  (STR ''x'' := storage_data.Value (Bytes [CHR 0xAA, CHR 0xBB, CHR 0xCC]))
                  (STR ''y'' := storage_data.Value (Bytes [CHR 0xAA, CHR 0xBB, CHR 0xCC]));
           storage = (λ_. undefined) (A1 := pstorage);
           state = emptyState(|Storage := storage|)
        in is_Normal (execute (do {
             assert_monad (equals_monad (contract_storeLookup (STR ''x'') []) (contract_storeLookup
(STR ''y'') []))
           }) state)"
```
⟨*proof*⟩

lemma
shows *"let pstorage =*
```
               ((λ_. undefined)
                  (STR ''x'' := storage_data.Value (Bytes [CHR 0xAA, CHR 0xBB, CHR 0xCC]))
                  (STR ''y'' := storage_data.Value (Bytes [CHR 0x00, CHR 0xAA, CHR 0xBB, CHR 0xCC]));
           storage = (λ_. undefined) (A1 := pstorage);
           state = emptyState(|Storage := storage|)
        in is_Exception (execute (do {
             assert_monad (equals_monad (contract_storeLookup (STR ''x'') []) (contract_storeLookup
(STR ''y'') []))
           }) state)"
```
⟨*proof*⟩

lemma
shows *"let pstorage =*
```
               ((λ_. undefined)
                  (STR ''x'' := storage_data.Value (Bytes [CHR 0xAA, CHR 0xBB, CHR 0xCC]))
                  (STR ''y'' := storage_data.Value (Bytes [CHR 0xAA, CHR 0xBB, CHR 0xCC, CHR 0x00]));
           storage = (λ_. undefined) (A1 := pstorage);
           state = emptyState(|Storage := storage|)
        in is_Exception (execute (do {
             assert_monad (equals_monad (contract_storeLookup (STR ''x'') []) (contract_storeLookup
(STR ''y'') []))
           }) state)"
```
⟨*proof*⟩

lemma
shows *"let pstorage =*
```
               (((λ_. undefined)
                  (STR ''x'' := storage_data.Value (Bytes [CHR 0x12, CHR 0x34, CHR 0x56]))
                  (STR ''y'' := storage_data.Value (Bytes [CHR 0xF0, CHR 0x87, CHR 0x4C]))
                  (STR ''z'' := storage_data.Value (Bytes [CHR 0x10, CHR 0x04, CHR 0x44]));
           storage = (λ_. undefined) (A1 := pstorage);
           state = emptyState(|Storage := storage|)
        in is_Normal (execute (do {
```

```
            assert_monad (equals_monad (contract_storeLookup (STR ''z'') []) (bytes_and_monad
(contract_storeLookup (STR ''x'') []) (contract_storeLookup (STR ''y'') [])))
          }) state)"
```
⟨*proof*⟩

context
includes *bit_operations_syntax*
begin

lemma *"word8_to_char ((char_to_word8 (CHR 0x56)) OR (char_to_word8 (CHR 0x4C))) = CHR 0x5E"*
⟨*proof*⟩

end

lemma
shows *"let pstorage =*
```
               (((λ_. undefined)
                  (STR ''x'' := storage_data.Value (Bytes [CHR 0x12, CHR 0x34, CHR 0x56])))
                  (STR ''y'' := storage_data.Value (Bytes [CHR 0xF0, CHR 0x87, CHR 0x4C])))
                  (STR ''z'' := storage_data.Value (Bytes [CHR 0xF2, CHR 0xB7, CHR 0x5E]));
          storage = (λ_. undefined) (A1 := pstorage);
          state = emptyState⦇Storage := storage⦈
       in is_Normal (execute (do {
            assert_monad (equals_monad (contract_storeLookup (STR ''z'') []) (bytes_or_monad
(contract_storeLookup (STR ''x'') []) (contract_storeLookup (STR ''y'') [])))
          }) state)"
```
⟨*proof*⟩

lemma
shows *"let pstorage =*
```
               (((λ_. undefined)
                  (STR ''x'' := storage_data.Value (Bytes [CHR 0x12, CHR 0x34, CHR 0x56])))
                  (STR ''y'' := storage_data.Value (Bytes [CHR 0xF0, CHR 0x87, CHR 0x4C])))
                  (STR ''z'' := storage_data.Value (Bytes [CHR 0xE2, CHR 0xB3, CHR 0x1A]));
          storage = (λ_. undefined) (A1 := pstorage);
          state = emptyState⦇Storage := storage⦈
       in is_Normal (execute (do {
            assert_monad (equals_monad (contract_storeLookup (STR ''z'') []) (bytes_xor_monad
(contract_storeLookup (STR ''x'') []) (contract_storeLookup (STR ''y'') [])))
          }) state)"
```
⟨*proof*⟩

lemma *"is_Normal (execute (do {*
```
       decl (TBytes 3) (STR ''x'');
       write (adata.Value (Bytes [CHR 0x00, CHR 0x00, CHR 0x00])) (STR ''y'');
       assert_monad (equals_monad (contract_stackLookup (STR ''x'') []) (contract_stackLookup (STR
''y'') []))
     }) emptyState)"
```
⟨*proof*⟩

lemma *"is_Normal (execute (do {*
```
       write (adata.Value (Bytes [CHR 0x12, CHR 0x34, CHR 0x56])) (STR ''x'');
       write (adata.Value (Bytes [CHR 0x12, CHR 0x34, CHR 0x56, CHR 0x00])) (STR ''y'');
```

```
        assert_monad (equals_monad (bytes_cast_monad 4 (contract_stackLookup (STR ''x'') []))
(contract_stackLookup (STR ''y'') []))
      }) emptyState)"
  ⟨proof⟩



lemma "is_Normal (execute (do {
        write (adata.Value (Bytes [CHR 0x12, CHR 0x34, CHR 0x56])) (STR ''x'');
        write (adata.Value (Bytes [CHR 0x12, CHR 0x34])) (STR ''y'');
        assert_monad (equals_monad (bytes_cast_monad 2 (contract_stackLookup (STR ''x'') []))
(contract_stackLookup (STR ''y'') []))
      }) emptyState)"
  ⟨proof⟩

lemma "is_Exception (execute (do {
         bytes_monad 1 []
      }) emptyState)"
  ⟨proof⟩

lemma "is_Normal (execute (do {
         bytes_monad 32 (array 32 CHR 0x00)
      }) emptyState)"
  ⟨proof⟩

lemma "is_Exception (execute (do {
         bytes_monad 33 (array 33 CHR 0x00)
      }) emptyState)"
  ⟨proof⟩

end
```

8 Applications

In this chapter we present three case studies in which we use Isabelle/Solidity to verify invariants for Solidity smart contracts.

```
theory Token
  imports Solidity_Main
begin
```

8.1 Token Contract (Token)

In the following we verify a simple token contract from https://www.isa-afp.org/entries/Solidity.html/.

8.1.1 Specification

```
abbreviation "balances ≡ STR ''balances''"
abbreviation "bal ≡ STR ''bal''"

contract Bank
  for balances: "SType.TMap (SType.TValue TAddress) (SType.TValue TSint)"

constructor payable
where
  "⟨skip⟩"

cfunction deposit external payable
where
  "balances [⟨sender⟩] ::=ₛ balances ~ₛ [⟨sender⟩] ⟨+⟩ ⟨value⟩" ,

cfunction withdraw external payable
where
  "do {
    bal :: TSint;
    bal [] ::= balances ~ₛ [⟨sender⟩];
    balances [⟨sender⟩] ::=ₛ ⟨sint⟩ 0;
    ⟨transfer⟩ ⟨sender⟩ (bal ~ [])
  }"

context bank
begin
  thm constructor_def
  thm deposit_def
  thm withdraw_def
end
```

8.1.2 Verifying an Invariant

```
abbreviation "SUMM x ≡ ∑ad∈UNIV. unat (valtype.uint (storage_data.vt (x (Address ad))))"

context Solidity
begin

lemma 1:
    fixes bal
  assumes "SUMM bal ≤ Balances s this"
      and "bal (Address msg_sender) = storage_data.Value (Uint y)"
      and "unat y + unat msg_value < 2^256"
    shows "(∑ad∈UNIV. unat (valtype.uint (storage_data.vt (if ad = msg_sender then storage_data.Value
(Uint (y + msg_value)) else bal (Address ad)))))
```

```
          ≤ Balances s this + unat msg_value"
```
⟨*proof*⟩

lemma *2:*
```
    fixes bal
  assumes "SUMM bal ≤ Balances s this"
      and "bal (Address msg_sender) = storage_data.Value (Uint y)"
    shows "(∑ ad ∈ UNIV. unat (valtype.uint (storage_data.vt (if ad = msg_sender then
storage_data.Value (Uint 0) else bal (Address ad)))))
         ≤ Balances s this + unat msg_value - unat y"
```
⟨*proof*⟩

lemma *3:*
```
    fixes bal
  assumes "SUMM bal ≤ Balances s this"
      and "bal (Address msg_sender) = storage_data.Value (Uint y)"
  shows "(∑ ad∈UNIV. unat (valtype.uint (storage_data.vt (if ad = msg_sender then storage_data.Value
(Uint 0) else bal (Address ad)))))
       ≤ Balances s this + unat msg_value"
```
⟨*proof*⟩

lemma *4:*
```
    fixes bal
  assumes "SUMM bal ≤ Balances s this"
      and "bal (Address msg_sender) = storage_data.Value (Uint y)"
      and " msg_sender = this"
  shows "(∑ ad∈UNIV. unat (valtype.uint (storage_data.vt (if ad = this then storage_data.Value (Uint
0) else bal (Address ad)))))
       ≤ Balances s this + unat msg_value"
```
⟨*proof*⟩

We need to create introduction and elimination rules for the invariant and add it to the wprules and wperule lists.

lemma (**in** *Solidity*) *bal_msg_sender:*
```
  fixes bal
  assumes "∀x. ∃y. bal x = storage_data.Value (Uint y)"
  obtains y where "bal (Address msg_sender) = storage_data.Value (Uint y)"
```
⟨*proof*⟩

Now we can start verifying the invariant. To this end our package provides a keyword invariant which takes a property as parameter and generates proof obligations.

end

invariant *sum_bal sb* **where**
```
    "∀x. (fst sb) balances = storage_data.Map x ⟶ (snd sb) ≥ SUMM x"
  for "Bank"
```

definition (**in** *Solidity*) *deposit_post* **where**
```
"deposit_post start_state return_value end_state ≡
  ∃x y. state.Storage start_state this STR ''balances'' = storage_data.Map x
  ∧ state.Storage end_state this STR ''balances'' = storage_data.Map y
  ∧ valtype.uint (storage_data.vt (y (Address msg_sender))) = valtype.uint (storage_data.vt (x (Address
msg_sender))) + msg_value"
```

declare (**in** *bank*) *sum_balI[wprules del]*

verification *sum_bal:*
```
  sum_bal
  "K True" "K (K (K True))"
  deposit "K True" "deposit_post" and
  withdraw "K True" "K (K (K True))"
  for "Bank"
```
⟨*proof*⟩

```
context bank_external
begin
  thm sum_bal
  thm vcond_def
end

end
theory Bank
  imports Solidity_Main
begin
```

8.2 Banking Contract (Bank)

8.2.1 Specification

```
abbreviation "balances ≡ STR ''balances''"
abbreviation "bal ≡ STR ''bal''"

contract Bank
  for balances: "SType.TMap (SType.TValue TAddress) (SType.TValue TSint)"

constructor
where
  "⟨skip⟩"

cfunction deposit external payable
where
  "balances [⟨sender⟩] ::=s balances ~s [⟨sender⟩] ⟨+⟩ ⟨value⟩" ,

cfunction reset
where
  "balances [⟨sender⟩] ::=s ⟨sint⟩ 0" ,

cfunction withdraw external
where
  "do {
    bal :: TSint;
    bal [] ::= balances ~s [⟨sender⟩];
    icall reset;
    ⟨transfer⟩ ⟨sender⟩ (bal ~ [])
  }"

context bank
begin
  thm constructor_def
  thm deposit_def
  thm withdraw_def
end
```

8.2.2 Verifying an Invariant

```
abbreviation "SUMM x ≡ ∑ad∈UNIV. unat (valtype.uint (storage_data.vt (x (Address ad))))"

context Solidity
begin

lemma 1:
    fixes bal
  assumes "SUMM bal ≤ Balances s this"
      and "bal (Address msg_sender) = storage_data.Value (Uint y)"
      and "unat y + unat msg_value < 2^256"
    shows "(∑ad∈UNIV. unat (valtype.uint (storage_data.vt (if ad = msg_sender then storage_data.Value
(Uint (y + msg_value)) else bal (Address ad)))))
```

```
          ≤ Balances s this + unat msg_value"
```
⟨*proof*⟩

lemma *21:*
```
    fixes bal bal'
  assumes "SUMM bal ≤ Balances s this"
      and "bal (Address msg_sender) = storage_data.Value (Uint y)"
      and "bal' (Address msg_sender) = storage_data.Value (Uint 0)"
      and "Balances s' this = Balances s this"
      and "⋀x. x ≠ msg_sender ⟹ bal' (Address x) = bal (Address x)"
    shows "SUMM bal' ≤ Balances s' this - unat y"
```
⟨*proof*⟩

lemma *22:*
```
    fixes bal bal'
  assumes "SUMM bal ≤ Balances s this"
      and "bal (Address msg_sender) = storage_data.Value (Uint y)"
      and "bal' (Address msg_sender) = storage_data.Value (Uint 0)"
      and "Balances s' this = Balances s this"
      and "⋀x. x ≠ msg_sender ⟹ bal' (Address x) = bal (Address x)"
    shows "SUMM bal' ≤ Balances s' this"
```
⟨*proof*⟩

lemma (**in** *Solidity*) *bal_msg_sender:*
```
  fixes bal
  assumes "∀x. ∃y. bal x = storage_data.Value (Uint y)"
  obtains y where "bal (Address msg_sender) = storage_data.Value (Uint y)"
```
⟨*proof*⟩

Now we can start verifying the invariant. To this end our package provides a keyword invariant which takes a property as parameter and generates proof obligations.

end

invariant *sum_bal sb* **where**
```
    "∀x. (fst sb) balances = storage_data.Map x ⟶ (snd sb) ≥ SUMM x"
  for "Bank"
```

abbreviation (**in** *Solidity*) *reset_post* **where**
```
"reset_post start_state return_value end_state ≡
  Balances start_state = Balances end_state ∧
  (∀mp. state.Storage start_state this balances = storage_data.Map mp ∧
  (∀y. ∃si. mp y = storage_data.Value (Uint si))
  ⟶ (∃mp'. state.Storage end_state this balances = storage_data.Map mp'
    ∧ mp' (Address msg_sender) = storage_data.Value (valtype.Uint 0)
    ∧ (∀x ≠ msg_sender. mp' (Address x) = mp (Address x))
    ∧ (∀y. ∃si. mp' y = storage_data.Value (Uint si))))"
```

declare (**in** *bank*) *sum_balI*[*wprules del*]

verification *sum_bal:*
```
  sum_bal
  "K True" "K (K (K True))"
  deposit "K True" "K (K (K True))" and
  withdraw "K True" "K (K (K True))" and
  reset "K True" reset_post
  for "Bank"
```
⟨*proof*⟩

context *bank_external*
begin
```
  thm sum_bal
  thm vcond_def
```
end

```
end
theory Casino
  imports Solidity_Main "HOL-Eisbach.Eisbach" "HOL-Eisbach.Eisbach_Tools"
begin
```

8.3 Casino Contract (Casino)

In the following we verify the Casino contract from the VerifyThis Long-Term Challenge: https://verifythis.github.io/02casino/ https://verifythis.github.io/ltc/02casino/.

8.3.1 Specification

In the following we describe the specification of the contract.

Method modifiers are formalized as abbreviations. They need to be formalized in the Solidity context to provide access to various contextual information.

```
abbreviation "state ≡ STR ''state''"
abbreviation "operator ≡ STR ''operator''"
abbreviation "player ≡ STR ''player''"
abbreviation "pot ≡ STR ''pot''"
abbreviation "hashedNumber ≡ STR ''hashedNumber''"
abbreviation "bet ≡ STR ''bet''"
abbreviation "guess ≡ STR ''guess''"
abbreviation "hashNum ≡ STR ''hashNum''"
abbreviation "secret ≡ STR ''secret''"
abbreviation "bet_old ≡ STR ''bet_old''"
abbreviation "secretNumber ≡ STR ''secretNumber''"
abbreviation "amount ≡ STR ''amount''"


abbreviation "IDLE ≡ 0"
abbreviation "GAME_AVAILABLE ≡ 1"
abbreviation "BET_PLACED ≡ 2"
abbreviation "HEADS ≡ 0"
abbreviation "TAILS ≡ 1"
abbreviation "CoinT ≡ SType.TValue TSint"
abbreviation "StateT ≡ SType.TValue TSint"
abbreviation "AddressT ≡ SType.TValue TAddress"
abbreviation "IntT ≡ SType.TValue TSint"
abbreviation "Bytes32T ≡ SType.TValue (TBytes 32)"


context Solidity
begin

abbreviation byOperator::"(unit, ex, 'a state) state_monad" where
  "byOperator ≡ assert Err (λs. storage_data.Value (valtype.Address msg_sender) = state.Storage s this
operator)"

abbreviation inState:: "'a valtype ⇒ (unit, ex, 'a state) state_monad" where
  "inState st ≡ assert Err (λs. state.Storage s this state = storage_data.Value st)"

abbreviation noActiveBet::"(unit, ex, ('a, 'b) state_scheme) state_monad" where
  "noActiveBet ≡ assert Err (λs. state.Storage s this state ≠ storage_data.Value (Uint 2))"

end
```

The contract can now be specified using the "contract" command. This command requires the following:

- A sequence of member variables
- A constructor
- A sequence of methods

```
contract Casino
  for state: StateT
  and operator: AddressT
  and player: AddressT
  and pot: IntT
  and hashedNumber: Bytes32T
  and bet: IntT
  and guess: CoinT

constructor payable
where
  "⟨skip⟩"

cfunction createGame external payable
  param hashNum: "SType.TValue (TBytes 32)"
where
  "do {
    byOperator;
    inState (valtype.Uint 0);
    hashedNumber [] ::=ₛ hashNum ~ [];
    state [] ::=ₛ ⟨sint⟩ 1
   }",

cfunction placeBet external payable
  param guess: "SType.TValue TSint"
where
  "do {
    inState (valtype.Uint 1);
    ⟨assert⟩ (⟨¬⟩ (⟨sender⟩ ⟨=⟩ operator ~ₛ []));
    ⟨assert⟩ (⟨value⟩ ⟨<⟩ (pot  ~ₛ []) ⟨∨⟩ ⟨value⟩ ⟨=⟩ (pot ~ₛ []));
    state [] ::=ₛ ⟨sint⟩ 2;
    player [] ::=ₛ ⟨sender⟩;
    bet [] ::=ₛ ⟨value⟩;
    guess [] ::=ₛ guess ~ []
  }",

cfunction decideBet external payable
  param secretNumber: IntT
where
  "do {
    byOperator;
    inState (Uint BET_PLACED);
    ⟨assert⟩ (hashedNumber ~ₛ [] ⟨=⟩ (⟨keccak256⟩ (secretNumber ~ [])));
    decl TSint secret;
    secret [] ::= IF ((secretNumber ~ []) ⟨%⟩ ⟨sint⟩ 2) ⟨=⟩ (⟨sint⟩ 0) THEN ⟨sint⟩ HEADS ELSE ⟨sint⟩
TAILS;
    IF (secret ~ []) ⟨=⟩ (guess ~ₛ []) THEN
      do {
        pot [] ::=ₛ ((pot ~ₛ []) ⟨-⟩ (bet ~ₛ []));
        bet [] ::=ₛ ⟨sint⟩ 0;
        ⟨transfer⟩ (player ~ₛ []) ((bet ~ₛ []) ⟨*⟩ (⟨sint⟩ 2))
      }
    ELSE
      do {
        pot [] ::=ₛ pot ~ₛ [] ⟨+⟩ bet ~ₛ [];
        bet [] ::=ₛ ⟨sint⟩ 0
      };
    state [] ::=ₛ ⟨sint⟩ IDLE
  }",

cfunction addToPot external payable
where
  "do {
    byOperator;
```

```
    pot [] ::=s pot ~s [] ⟨+⟩ ⟨value⟩
  }",

cfunction removeFromPot external payable
  param amount: "SType.TValue TSint"
where
  "do {
    byOperator;
    noActiveBet;
    pot [] ::=s ((pot ~s []) ⟨-⟩ (amount ~ []));
    ⟨transfer⟩ (operator ~s []) (amount ~ [])
  }"

thm casino.addToPot_def

invariant pot_balance sb where
    "(fst sb state = storage_data.Value (Uint BET_PLACED)
      ⟶ snd sb ≥ unat (valtype.uint (storage_data.vt (fst sb pot))) + unat (valtype.uint
(storage_data.vt (fst sb bet)))
        ∧ valtype.uint (storage_data.vt (fst sb bet)) ≤ valtype.uint (storage_data.vt (fst sb pot))) ∧
    (fst sb state ≠ storage_data.Value (Uint BET_PLACED)
      ⟶ snd sb ≥ unat (valtype.uint (storage_data.vt (fst sb pot))))"
  for "casino"
```

8.4 Verifying an Invariant (Casino)

We start by verifying an invariant regarding the relationship between pot and balance. To this end we need to add type information to the invariant. Note that an invariant is formalized as a predicate over the contracts state and balance.

We need to create introduction and elimination rules for the invariant and add it to the wprules and wperule lists.

Now we can start verifying the invariant. To this end our package provides a keyword invariant which takes a property as parameter and generates proof obligations.

```
context casino
begin

lemma sym2:
  assumes "storage_data.Value v = state.Storage s this x"
    shows "state.Storage s this x = storage_data.Value v"
  ⟨proof⟩

lemma kdequals_true_dest[wpdrules]:
  assumes "kdequals a b = Some (rvalue.Value (Bool True))"
  shows "a = b"
  ⟨proof⟩


lemma kdequals_false_dest[wpdrules]:
assumes "kdequals a b = Some (rvalue.Value (Bool False))"
shows "a ≠ b"
  ⟨proof⟩

lemma kdnot_dest[wpdrules]:
  assumes "kdnot ya = Some (rvalue.Value (Bool True))"
  shows "ya=rvalue.Value (Bool False)"
  ⟨proof⟩

lemma vt_or_dest:
  assumes "lift_value_binary vtor ye yg = Some (rvalue.Value (Bool True))"
  shows "ye = rvalue.Value (Bool True) ∨ yg = rvalue.Value (Bool True)"
 ⟨proof⟩
```

lemma *kdless_dest[wpdrules]:*
assumes *"kdless (rvalue.Value (Uint x)) (rvalue.Value (Uint y)) = Some (rvalue.Value (Bool True))"*
shows *"x < y"*
⟨*proof*⟩

lemma *kdequals_bool_sint[wpsimps]: "kdequals (rvalue.Value (Bool x)) (rvalue.Value (Uint y)) = None"*
⟨*proof*⟩

lemma *unat_add_imp:*
assumes *"(unat (x::256 word) + unat y < 115792089237316195423570985008687907853269984665640564039457584007913129*
shows *"(unat (x + y) = unat x + unat y)"* ⟨*proof*⟩

lemma *unat_mult_imp:*
assumes *"(unat (x::256 word) * unat y < 115792089237316195423570985008687907853269984665640564039457584007913129*
shows *"(unat (x * y) = unat x * unat y)"* ⟨*proof*⟩

lemma *unat_sub_imp:*
assumes *"(y::256 word) ≤ (x::256 word)"*
shows *"(unat (x - y) = unat x - unat y)"* ⟨*proof*⟩

lemma *no_plus_overflow_leq:*
assumes *a:"(x ::256 word) ≤ y"*
and *b: "unat y + unat z < 2 ^ 256"*
shows *"x ≤ y + z"* ⟨*proof*⟩

lemma *111:*
fixes *pot*
assumes *"unat pot ≤ Balances s this"*
and **: "x ≤ pot"*
shows *"unat (pot - x) ≤ Balances s this + unat msg_value - unat x"*
⟨*proof*⟩

lemma *222: "⋀pot. unat pot ≤ Balances s this ⟹*
x ≤ pot ⟹
unat (pot - x) ≤ Balances s this + unat msg_value"
⟨*proof*⟩

end

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

lemma **(in** *Contract)* *wp_assign_storage111[wperules]:*
assumes *"rvalue.Value v=y"*
and *"wp (storage_update_monad [] is (K (storage_data.Value v)) i) P E s"*
shows *"wp (assign_storage i is y) P E s"*
⟨*proof*⟩

lemma *notwp[wperules]: "¬ wp m P E s ⟹ wp m P E s ⟹ R"* ⟨*proof*⟩

abbreviation(in *Contract)* *createGame_post* **where**
"createGame_post hN start_state return_value end_state ≡
state.Storage end_state this STR ''state'' = storage_data.Value (Uint 1)"

abbreviation(in *Contract)* *placeBet_post* **where**
"placeBet_post hn start_state return_value end_state ≡
state.Storage end_state this STR ''state'' = storage_data.Value (Uint 2)"

abbreviation(in *Contract)* *decideBet_post* **where**
"decideBet_post hn start_state return_value end_state ≡
state.Storage end_state this STR ''state'' = storage_data.Value (Uint 0)"

declare(in *casino)* *pot_balanceI[wprules del]*

```
verification pot_balance:
  pot_balance
  "(K True)" "K (K (K True))"
  "createGame" "(K (K True))" "createGame_post" and
  "placeBet" "(K (K True))" "placeBet_post" and
  "decideBet" "(K (K True))" "decideBet_post" and
  "addToPot" "(K True)" "K (K (K True))" and
  "removeFromPot" "(K (K True))" "K (K (K (K True)))"
  for "casino"
⟨proof⟩

context casino_external
begin
  thm pot_balance
  thm vcond_def
end

end
theory Voting
  imports Solidity_Main
begin
```

8.5 Voting Contract (Voting)

In the following we verify the Voting contract from the official Solidity documentation: https://docs.soliditylang.org/en/v0.8.25/solidity-by-example.html#voting.

```
lemma kdequals_true[wpdrules]:
  assumes "kdequals (rvalue.Value (Uint w)) (rvalue.Value (Uint x)) = Some (rvalue.Value (Bool True))"
  shows "w = x"
  ⟨proof⟩
```

8.5.1 Specification

```
abbreviation TT::"((String.literal ⇒ 'a storage_data) × nat) ⇒ bool" where "TT a ≡ True"

context Solidity
begin

abbreviation Voter where "Voter ≡ storage_data.Array [storage_data.Value (Uint 0),storage_data.Value
(Bool False),storage_data.Value (Address null),storage_data.Value (Uint 0::'a valtype)]"
abbreviation "weight ≡ return (rvalue.Value ((Uint 0)::'a valtype))"
abbreviation "voted ≡ 1::nat"
abbreviation "sdelegate ≡ 2::nat"
abbreviation "svote ≡ 3::nat"

abbreviation "Proposal name voteCount ≡ storage_data.Array [name::'a valtype storage_data, voteCount]"
abbreviation "name ≡ 0::nat"
abbreviation "voteCount ≡ 1::nat"

end
```

The contract can now be specified using the "contract" command. This command requires the following:

- A sequence of member variables
- A constructor
- A sequence of methods

```
context Solidity
begin
```

```
abbreviation "SUMM (x::('a valtype ⇒ 'a valtype storage_data)) ≡ ∑ad|∃ v. x (Address ad) =
storage_data.Array v ∧ valtype.bool (storage_data.vt (v ! 1)). (THE y. ∃ v. x (Address ad) =
storage_data.Array v ∧ y=unat (valtype.uint (storage_data.vt (v ! 0))))"
abbreviation "SUMM2 (x::'a valtype storage_data list) ≡ ∑i<length x. (THE y. ∃ p. x ! i =
storage_data.Array p ∧ y=unat (valtype.uint (storage_data.vt (p ! 1))))"

definition inv':: "(id ⇒ 'a valtype storage_data) ⇒ bool"
  where "inv' s ≡ (∀ x y. s (STR ''voters'') = storage_data.Map x
                       ∧ s (STR ''proposals'') = storage_data.Array y
                            ⟶ (SUMM2 y ≤ SUMM x))"

end

contract Ballot
  for "STR ''chairperson''": "SType.TValue TAddress"
  and "STR ''voters''": "SType.TMap (SType.TValue TAddress) (SType.TEnum [SType.TValue TSint,
SType.TValue TBool, SType.TValue TAddress, SType.TValue TSint])"
  and "STR ''proposals''": "SType.DArray (SType.TEnum [SType.TValue (TBytes 32), SType.TValue TSint])"

constructor payable
  memory "STR ''proposalNames''": "SType.DArray (SType.TValue (TBytes 32))"
where
  "do {
    assign_storage_monad (STR ''chairperson'') [] sender_monad;
    assign_storage_monad (STR ''voters'') [stackLookup (STR ''chairperson'') [], weight] (sint_monad 1);
    init (Uint 0) (STR ''i'');
    while_monad (less_monad (stackLookup (STR ''i'') []) (arrayLength (STR ''proposalNames'') []))
      (do {
        allocate (STR ''proposals'') [] (Proposal (storage_data.Value (Bytes (array 32 (CHR 0x00))))
(storage_data.Value (Uint 0)));
        assign_storage_monad (STR ''proposals'') [storeArrayLength (STR ''proposals'') [], sint_monad 0]
(stackLookup (STR ''proposalNames'') [stackLookup (STR ''i'') []]);
        assign_storage_monad (STR ''proposals'') [storeArrayLength (STR ''proposals'') [], sint_monad 1]
(sint_monad 0);
        assign_stack_monad (STR ''i'') [] (plus_monad (stackLookup (STR ''i'') []) (sint_monad 1))
      })
   }"

cfunction giveRightToVote external payable
  param "STR ''voter''": "SType.TValue TAddress"
where
  "do {
    require_monad (equals_monad sender_monad (storeLookup (STR ''chairperson'') []));
    require_monad (not_monad (storeLookup (STR ''voters'') [stackLookup (STR ''voter'') [], sint_monad
1]));
    require_monad (equals_monad (storeLookup (STR ''voters'') [stackLookup (STR ''voter'') [],
sint_monad 0]) (sint_monad 0));
    assign_storage_monad (STR ''voters'') [stackLookup (STR ''voter'') [], sint_monad 0] (sint_monad 1)
  }",

cfunction delegate external payable
  param "STR ''to''": "SType.TValue TAddress"
where
  "do {
    sdecl (SType.TEnum [SType.TValue TSint, SType.TValue TBool, SType.TValue TAddress, SType.TValue
TSint]) (STR ''sender'');
    assign_stack_monad (STR ''sender'') [] (storeLookup (STR ''voters'') [sender_monad]);
    require_monad (not_monad (equals_monad (stackLookup (STR ''sender'') [sint_monad 0]) (sint_monad
0)));
    require_monad (not_monad (stackLookup (STR ''sender'') [sint_monad 1]));
    require_monad (not_monad (equals_monad (stackLookup (STR ''to'') []) sender_monad));
    while_monad (equals_monad (storeLookup (STR ''voters'') [stackLookup (STR ''to'') [], sint_monad 2])
(address_monad null))
      (do {
```

```
        assign_stack_monad (STR ''to'') [] (storeLookup (STR ''voters'') [stackLookup (STR ''to'') [],
sint_monad 2]);
        require_monad (not_monad (equals_monad (stackLookup (STR ''to'') []) sender_monad))
      });
    sdecl (SType.TEnum [SType.TValue TSint, SType.TValue TBool, SType.TValue TAddress, SType.TValue
TSint]) (STR ''delegate_'');
    assign_stack_monad (STR ''delegate_'') [] (storeLookup (STR ''voters'') [stackLookup (STR ''to'')
[]]);
    require_monad (not_monad (less_monad (stackLookup (STR ''delegate_'') [sint_monad 0]) (sint_monad
1)));
    assign_stack_monad (STR ''sender'') [sint_monad 1] (true_monad);
    assign_stack_monad (STR ''delegate_'') [sint_monad 2] (stackLookup (STR ''to'') []);
    require_monad (not_monad (stackLookup (STR ''delegate_'') [sint_monad 1]));
    cond_monad (stackLookup (STR ''delegate_'') [sint_monad 1])
      (assign_storage_monad (STR ''proposals'') [stackLookup (STR ''delegate_'') [sint_monad 3],
sint_monad 1] (plus_monad_safe (storeLookup (STR ''proposals'') [stackLookup (STR ''delegate_'')
[sint_monad 3], sint_monad 1]) (stackLookup (STR ''sender'') [sint_monad 0])))
      (assign_stack_monad (STR ''delegate_'') [sint_monad 0] (plus_monad_safe (stackLookup (STR
''delegate_'') [sint_monad 0]) (stackLookup (STR ''sender'') [sint_monad 0])))
  }",

cfunction vote external payable
  param "STR ''proposal''": "SType.TValue TSint"
where
  "do {
    sdecl (SType.TEnum [SType.TValue TSint, SType.TValue TBool, SType.TValue TAddress, SType.TValue
TSint]) (STR ''sender'');
    assign_stack_monad (STR ''sender'') [] (storeLookup (STR ''voters'') [sender_monad]);
    require_monad (not_monad (equals_monad (stackLookup (STR ''sender'') [sint_monad 0]) (sint_monad
0)));
    require_monad (not_monad (stackLookup (STR ''sender'') [sint_monad 1]));
    assign_stack_monad (STR ''sender'') [sint_monad 1] (true_monad);
    assign_stack_monad (STR ''sender'') [sint_monad 4] (stackLookup (STR ''proposal'') []);
    assign_storage_monad (STR ''proposals'') [stackLookup (STR ''proposal'') [], sint_monad 1]
(plus_monad_safe (storeLookup (STR ''proposals'') [stackLookup (STR ''proposal'') [], sint_monad 1])
(stackLookup (STR ''sender'') [sint_monad 0]))
  }",

cfunction winningProposal external payable
  param "STR ''winningProposalu''": "SType.TValue TSint"
where
  "do {
    init (Uint 0) (STR ''winningVoteCount'');
    init (Uint 0) (STR ''p'');
    while_monad (less_monad (stackLookup (STR ''p'') []) (storeArrayLength (STR ''proposals'') []))
      (do {
        cond_monad (less_monad (stackLookup (STR ''winningVoteCount'') []) (storeLookup (STR
''proposals'') [stackLookup (STR ''p'') [], sint_monad 1]))
          (do {
            assign_stack_monad (STR ''winningVoteCount'') [] (storeLookup (STR ''proposals'')
[stackLookup (STR ''p'') [], sint_monad 1]);
            assign_stack_monad (STR ''winningProposalu'') [] (stackLookup (STR ''p'') [])
          })
          (skip_monad)
      })
  }"

invariant sum_votes s
  where "inv' (fst s)"
  for "Ballot"

context ballot
begin
```

lemma *obtain_props:*
assumes *"∀ya < length a. ∃ema. a ! ya = storage_data.Array ema ∧ (∃bt sib. ema = [storage_data.Value (Bytes bt), storage_data.Value (Uint sib)])"*
and *"unat ya < length a"*
obtains *bt sib* **where** *"a ! unat ya = storage_data.Array [storage_data.Value (Bytes bt), storage_data.Value (Uint sib)]"*
⟨*proof*⟩

lemma *obtain_voters:*
assumes *"∀x. ∃em. voters x = storage_data.Array em ∧ (∃w t d v. em = [storage_data.Value (Uint w), storage_data.Value (Bool t), storage_data.Value (Address d), storage_data.Value (Uint v)])"*
obtains *w t d v* **where** *"voters x = storage_data.Array [storage_data.Value (Uint w), storage_data.Value (Bool t), storage_data.Value (Address d), storage_data.Value (Uint v)]"*
⟨*proof*⟩

lemma *kdequals_true[wpdrules]:*
assumes *"kdequals (rvalue.Value (Address x)) (rvalue.Value (Address y)) = Some (rvalue.Value (Bool True))"*
shows *"x = y"*
⟨*proof*⟩

lemma *kdequals_false:*
assumes *"kdequals (rvalue.Value (Address x)) (rvalue.Value (Address y)) = Some (rvalue.Value (Bool False))"*
shows *"x ≠ y"*
⟨*proof*⟩

lemma *kdnot[wpdrules]:*
assumes *"kdnot (rvalue.Value (Bool t)) = Some (rvalue.Value (Bool True))"*
shows *"¬ t"*
⟨*proof*⟩

lemma *inv_0:*
assumes *"state.Storage s this STR ''voters'' = Map voters"*
and *"state.Storage s this STR ''proposals'' = storage_data.Array proposals"*
and *"inv' (state.Storage s this)"*
and *"voters (Address x) = storage_data.Array [storage_data.Value (Uint 0), storage_data.Value (Bool False), storage_data.Value (Address d), storage_data.Value (Uint v)]"*
shows *"inv' (state.Storage*
(storage_update STR ''voters'' (Map (voters (Address x := storage_data.Array [storage_data.Value (Uint 1), storage_data.Value (Bool False), storage_data.Value (Address d), storage_data.Value (Uint v)])))
(stack_update STR ''voter'' (kdata.Value (Address x))
(balance_update (Balances s this + unat msg_value)
s(|Stack := {$$}, Memory := [], Calldata := {$$}|))) this)"
⟨*proof*⟩

lemma *inv_1:*
assumes *"state.Storage s this STR ''voters'' = Map voters"*
and *"state.Storage s this STR ''proposals'' = storage_data.Array proposals"*
and *"inv' (state.Storage s this)"*
and *"voters (Address msg_sender) = storage_data.Array [storage_data.Value (Uint w), storage_data.Value (Bool False), storage_data.Value (Address d), storage_data.Value (Uint v)]"*
and *"state.Storage sa this STR ''proposals'' = storage_data.Array proposals"*
and *"xa ≠ msg_sender"*
and *"voters (Address xa) = storage_data.Array [storage_data.Value (Uint wb), storage_data.Value (Bool False), storage_data.Value (Address da), storage_data.Value (Uint va)]"*
shows *"inv' (state.Storage*
(storage_update STR ''voters'' (Map (voters
(Address msg_sender := storage_data.Array [storage_data.Value (Uint w), storage_data.Value (Bool True), storage_data.Value (Address d), storage_data.Value (Uint v)],
Address xa := storage_data.Array [storage_data.Value (Uint (wb + w)),

```
storage_data.Value (Bool False), storage_data.Value (Address xa), storage_data.Value (Uint va))))
          (storage_update STR ''voters'' (Map (voters
                  (Address msg_sender := storage_data.Array [storage_data.Value (Uint w),
storage_data.Value (Bool True), storage_data.Value (Address d), storage_data.Value (Uint v)],
                   Address xa := storage_data.Array [storage_data.Value (Uint wb), storage_data.Value
(Bool False), storage_data.Value (Address xa), storage_data.Value (Uint va)])))
          (storage_update STR ''voters'' (Map (voters
                (Address msg_sender := storage_data.Array [storage_data.Value (Uint w),
storage_data.Value (Bool True), storage_data.Value (Address d), storage_data.Value (Uint v)])))
          (stack_update STR ''delegate_'' (kdata.Storage (Some (|Location=STR ''voters'', Offset=
[Address xa]|))) (stack_update STR ''delegate_'' (kdata.Storage None)
          sa))))) this)"
  ⟨proof⟩

lemma wp_assign_storage[wprules]:
  assumes "⋀y. updateStore is (K (storage_data.Value v)) (state.Storage s this i) = Some y ⟹ P
Empty (storage_update i y s)"
      and "updateStore ([] @ is) (K (storage_data.Value v)) (state.Storage s this i) = None ⟹ E Err
s"
    shows "wp (assign_storage i is (rvalue.Value v)) P E s"
  ⟨proof⟩

lemma slookup_some[wpdrules]:
  assumes "proposals $ x >>= slookup [Uint 1] = Some yp"
  shows "x < length proposals ∧ slookup [Uint 1] (proposals ! x) = Some yp"
  ⟨proof⟩

lemma kdplus_safe_storage_none[wpsimps]:
  shows "kdplus_safe x (rvalue.Storage p) = None"
  ⟨proof⟩

lemma kdplus_safe_storage_some_false[wpdrules]:
  assumes "kdplus_safe x (rvalue.Storage p) = Some a"
  shows "False"
  ⟨proof⟩

lemma kdequals2[wpdrules]:
  assumes "kdequals x y = Some (rvalue.Value (Bool True))"
  shows "x = y"
  ⟨proof⟩

lemma kdequals3[wpdrules]:
  assumes "kdequals x y = Some (rvalue.Value (Bool False))"
  shows "¬ x = y"
  ⟨proof⟩

lemma wp_less_monad[wprules]:
  assumes "wp lm (λa. wp (rm >>= (λrv. option Err (K (kdless a rv)) >>= return)) P E) E s"
  shows "wp (less_monad lm rm) P E s"
  ⟨proof⟩

lemma kdnot_true[wpdrules]:
  assumes "kdnot y = Some (rvalue.Value (Bool True))"
  shows "y=rvalue.Value (Bool False)"
  ⟨proof⟩

lemma list_update_safe_simps1[wpsimps]:
  assumes "i < length xs"
  shows "list_update_safe xs i a = Some (list_update xs i a)"
  ⟨proof⟩
end

lemma notwp[wperules]: "¬ wp m P E s ⟹ wp m P E s ⟹ R" ⟨proof⟩
method solve methods m = (m ; fail)
```

```
context ballot
begin

declare sum_votesI[wprules del]

end

verification sum_votes:
  "sum_votes"
  "K (K True)" "K (K (K (K True)))"
  "giveRightToVote" "K (K True)" "K (K (K (K True)))" and
  "delegate" "K (K True)" "K (K (K (K True)))" and
  "vote" "K (K True)" "K (K (K (K True)))" and
  "winningProposal" "K (K True)" "K (K (K (K True)))"
  for "Ballot"
⟨proof⟩

context ballot_external
begin
  thm sum_votes
  thm vcond_def
end

end
theory SimpleAuction
  imports Solidity_Main
begin
```

8.6 Simple Auction Contract (SimpleAuction)

```
abbreviation "SUMM x ≡ ∑ad∈UNIV. unat (valtype.uint (storage_data.vt (x (Address ad))))"
```

In the following we verify the Blind Auction contract from the official Solidity documentation: https://docs.soliditylang.org/en/v0.8.25/solidity-by-example.html#blind-auction.

8.6.1 Specification

The contract can now be specified using the "contract" command. This command requires the following:

- A sequence of member variables
- A constructor
- A sequence of methods

```
contract simpleauction
  for "STR ''Beneficiary''" : "SType.TValue TAddress"
  and "STR ''auctionEndTime''" :  "SType.TValue TSint"
  and "STR ''highestBidder''" : "SType.TValue TAddress"
  and "STR ''highestBid''" : "SType.TValue TSint"
  and "STR ''pendingReturns''" : "SType.TMap (SType.TValue TAddress) (SType.TValue TSint)"
  and "STR ''ended''" : "SType.TValue TBool"

constructor payable
param "STR ''biddingTime''": "SType.TValue TSint"
and  "STR ''beneficiaryAddress''": "SType.TValue TAddress"
  where
  "do {
      assign_storage_monad (STR ''Beneficiary'') [] (stackLookup (STR ''beneficiaryAddress'') []);
      assign_storage_monad (STR ''auctionEndTime'') []  (plus_monad_safe (stackLookup (STR
''biddingTime'') []) (block_timestamp_monad))
      }
   "
```

```
cfunction bid external payable
where
"do
{
assert_monad (not_monad (less_monad  (storeLookup (STR ''auctionEndTime'') [])
(block_timestamp_monad)));
assert_monad (less_monad  (storeLookup (STR ''highestBid'') []) (value_monad));
 cond_monad (not_monad (equals_monad (storeLookup (STR ''highestBid'') []) (sint_monad 0)))
  (do {
      init (Uint 0) (STR ''temp_stack_variable'');
      assign_stack_monad  (STR ''temp_stack_variable'') [] (plus_monad_safe (storeLookup (STR
''pendingReturns'') [storeLookup (STR ''highestBidder'') []]) ((storeLookup (STR ''highestBid'') [])));
      assign_storage_monad  (STR ''pendingReturns'') [storeLookup (STR ''highestBidder'') []]
(stackLookup (STR ''temp_stack_variable'') [])
        }) (skip_monad);
      assign_storage_monad (STR ''highestBidder'') [] (sender_monad);
      assign_storage_monad (STR ''highestBid'') [] (value_monad)
}",

cfunction withdraw external payable
where
  "do {
  init (Uint 0) (STR ''amount'');
  assign_stack_monad (STR ''amount'') [] (storeLookup (STR ''pendingReturns'') [sender_monad] );
  cond_monad (less_monad (sint_monad 0) (stackLookup (STR ''amount'') []))
  (do {
      assign_storage_monad (STR ''pendingReturns'') [sender_monad] (sint_monad 0);
      transfer_monad (sender_monad) (stackLookup  (STR ''amount'') [])
   }) (skip_monad)
   }",

cfunction auctionEnded external payable
where
  "do {
  assert_monad ((not_monad (less_monad  (block_timestamp_monad) (storeLookup (STR ''auctionEndTime'')
[]))));
  assert_monad (equals_monad (storeLookup (STR ''ended'') []) (false_monad));
  assign_storage_monad (STR ''ended'') [] (true_monad);
   transfer_monad (storeLookup (STR ''Beneficiary'') []) (storeLookup  (STR ''highestBid'') [])
         }"
```

8.7 Verifying an invariant (SimpleAuction)

```
invariant pr_less_Balance s
  where "(∀x y e.
            (fst s) (STR ''ended'') = storage_data.Value (Bool e) ∧
            (fst s) (STR ''pendingReturns'') = storage_data.Map x ∧
            (fst s) (STR ''highestBid'') =  storage_data.Value (Uint y)
          ⟶ (if e = False then ((snd s) ≥ SUMM x + unat y) else  (snd s) ≥ SUMM x ))"
  for "simpleauction"

thm simpleauction.pr_less_BalanceI

lemma kdequals_true[wpdrules]:
  assumes "kdequals (rvalue.Value (Address x)) (rvalue.Value (Address y)) = Some (rvalue.Value (Bool
True))"
  shows "x = y"
  ⟨proof⟩

lemma kdequals_false:
  assumes "kdequals (rvalue.Value (Address x)) (rvalue.Value (Address y)) = Some (rvalue.Value (Bool
False))"
  shows "x ≠ y"
```

⟨*proof*⟩

lemma *kdnot[wpdrules]:*
assumes *"kdnot (rvalue.Value (Bool t)) = Some (rvalue.Value (Bool True))"*
shows *"¬ t"*
⟨*proof*⟩

lemma *kdnot2[wpdrules]:*
assumes *"kdnot x = Some (rvalue.Value (Bool t))"*
shows *"x = (rvalue.Value (Bool (¬ t)))"*
⟨*proof*⟩

lemma *kdequals_true_arg_rvalue_true:*
assumes *"kdequals yb (rvalue.Value (Bool True)) = Some (rvalue.Value (Bool True))"*
shows *"yb = rvalue.Value (Bool True)"*
⟨*proof*⟩

lemma *rvalue_equal_kdequals_true:*
assumes *"kdequals y x = Some (rvalue.Value (Bool True))"*
shows *"y = x"*
⟨*proof*⟩

lemma *kdequals_true_arg_rvalue_false:*
assumes *"kdequals (rvalue.Value (Bool xe)) (rvalue.Value (Bool False)) = Some (rvalue.Value (Bool True))"*
shows *"xe = False"* ⟨*proof*⟩

lemma *kdplus_safe_storage_none[wpsimps]:*
shows *"kdplus_safe x (rvalue.Storage p) = None"*
⟨*proof*⟩

lemma *kdplus_safe_storage_some_false[wpdrules]:*
assumes *"kdplus_safe x (rvalue.Storage p) = Some a"*
shows *"False"*
⟨*proof*⟩

lemma *storage_data_value_is_value [wpsimps]:*
assumes *"storage_data.Value (Uint xc) = y"*
shows *"storage_data.is_Value y"*
⟨*proof*⟩

lemma *kdplussafe_sint [wpsimps]:*
assumes *"∀a. ∃y. x a = storage_data.Value (Uint y)"*
obtains *y* **where** *"x a = storage_data.Value (Uint y)"* ⟨*proof*⟩

8.8 Verifying an Invariant (SimpleAuction)

lemma *notwp[wperules]: "¬ wp m P E s ⟹ wp m P E s ⟹ R"* ⟨*proof*⟩

lemma *kdequals_rvalue_false_larg_inf:* **assumes** *"kdequals yi (rvalue.Value (Bool i)) = Some (rvalue.Value (Bool False))"*
shows *"yi = (rvalue.Value (Bool (¬ i)))"*
⟨*proof*⟩

lemma *add_right_lessineq:*
assumes *"xb (Address msg_sender) = storage_data.Value (Uint y)"*
and *" (∑ ad∈UNIV. unat (valtype.uint (storage_data.vt (xb (Address ad))))) + unat xd ≤ Balances s this"*
shows *"(∑ ad∈UNIV. unat (valtype.uint (storage_data.vt (if ad = this then storage_data.Value (Uint (0)) else xb (Address ad)))))*
+ unat xd ≤ Balances s this + unat msg_value"
⟨*proof*⟩

lemma *sub_right_lessineq:*
assumes *"xb (Address msg_sender) = storage_data.Value (Uint y)"*
and *" (∑ ad∈UNIV. unat (valtype.uint (storage_data.vt (xb (Address ad))))) + unat xd ≤ Balances s this"*
shows *" (∑ ad∈UNIV. unat (valtype.uint (storage_data.vt (if ad = msg_sender then storage_data.Value (Uint 0) else xb (Address ad))))) + unat xd ≤ Balances s this + unat msg_value - unat y"*
⟨*proof*⟩

lemma *selfcall_balance_ineq:*
assumes *"xb (Address msg_sender) = storage_data.Value (Uint y)"*
and *" (∑ ad∈UNIV. unat (valtype.uint (storage_data.vt (xb (Address ad))))) ≤ Balances s msg_sender"*
and *"msg_sender = this"*
shows *" (∑ ad∈UNIV. unat (valtype.uint (storage_data.vt (if ad = msg_sender then storage_data.Value (Uint 0) else xb (Address ad))))) ≤ Balances s msg_sender + unat msg_value"*
⟨*proof*⟩

lemma *notselfcall_balance_ineq:*
assumes *"(∑ ad∈UNIV. unat (valtype.uint (storage_data.vt (xb (Address ad))))) ≤ Balances s this"*
and *"xb (Address msg_sender) = storage_data.Value (Uint ya)"*
and *"unat ya ≤ Balances s this + unat msg_value"*
and *"this ≠ msg_sender"*
shows *" (∑ ad∈UNIV. unat (valtype.uint (storage_data.vt (if ad = msg_sender then storage_data.Value (Uint 0) else xb (Address ad))))) ≤ Balances s this + unat msg_value - unat ya"*
⟨*proof*⟩

lemma *sum_def_spec:*
assumes *"SUMM xb + unat xd ≤ Balances s this"*
and *"xb (Address xc) = storage_data.Value (Uint yf)"*
and *"unat yf + unat xd < 2^256"*
shows *" (∑ ad∈UNIV. unat (valtype.uint (storage_data.vt (if ad = xc then storage_data.Value (Uint (yf + xd)) else xb (Address ad))))) ≤ Balances s this"*
⟨*proof*⟩

lemma *highestbid_update:*
assumes *"SUMM xb ≤ Balances s this"*
and *"xb (Address xc) = storage_data.Value (Uint yf)"*
and *"unat yf + unat xd < 2^256"*
and *"xd < msg_value"*
shows *" (∑ ad∈UNIV. unat (valtype.uint (storage_data.vt (if ad = xc then storage_data.Value (Uint (yf + xd)) else xb (Address ad))))) ≤ Balances s this + unat msg_value"*
⟨*proof*⟩

abbreviation (**in** *Solidity*) *post_const* **where**
"post_const (bidding_time::256 word) beneficiary start_state return_value end_state ≡
state.Storage end_state this STR ''Beneficiary'' = storage_data.Value (Address beneficiary) ∧
state.Storage end_state this STR ''auctionEndTime'' = storage_data.Value (Uint (timestamp + bidding_time))"

abbreviation (**in** *Solidity*) *post_bid* **where**
"post_bid start_state return_value end_state ≡
state.Storage end_state this STR ''highestBidder'' = storage_data.Value (Address msg_sender) ∧
state.Storage end_state this STR ''highestBid'' = storage_data.Value (Uint msg_value) ∧
¬ (state.Storage start_state this STR ''highestBid'' = storage_data.Value (Uint 0))
⟶ (∀ mo mn b h prn pro.
state.Storage start_state this STR ''pendingReturns'' = storage_data.Map mo ∧
state.Storage end_state this STR ''pendingReturns'' = storage_data.Map mn ∧

```
        state.Storage start_state this STR ''highestBidder'' = storage_data.Value (Address b) ∧
        state.Storage start_state this STR ''highestBid'' = storage_data.Value (Uint h) ∧
        mn (Address b) = storage_data.Value (Uint prn) ∧
        mo (Address b) = storage_data.Value (Uint pro)
        ⟶ prn = pro + h)"
```

abbreviation (**in** *Contract*) *post_bid2* **where**
"post_bid2 start_state return_value end_state ≡ True"

lemma *kdless_true_arg_true:* **assumes** *"kdless (rvalue.Value (Uint xd)) (rvalue.Value (Uint msg_value))*
= Some (rvalue.Value (Bool True))"
shows *"xd < msg_value"*
⟨*proof*⟩

declare(**in** *simpleauction*) *pr_less_BalanceI[wprules del]*

verification *pr_less_Balance:*
pr_less_Balance
"K (K (K True))" "post_const"
"bid" "K True" "post_bid" **and**
"withdraw" "K True" "K (K (K True))" **and**
"auctionEnded" "K True" "K (K (K True))"
for *"simpleauction"*

⟨*proof*⟩

end

Bibliography

[Marmsoler and Brucker(2022)] D. Marmsoler and A. D. Brucker. Isabelle/solidity: A deep embedding of solidity in isabelle/hol. *Archive of Formal Proofs*, July 2022. ISSN 2150-914x. https://isa-afp.org/entries/Solidity.html, Formal proof development.

[Marmsoler et al.(2024)Marmsoler, Ahmed, and Brucker] D. Marmsoler, A. Ahmed, and A. D. Brucker. Secure smart contracts with Isabelle/Solidity. In *International Conference on Software Engineering and Formal Methods*, pages 162–181. Springer, 2024.