

Isabelle/Solidity

A shallow Embedding of Solidity in Isabelle/HOL

Diego Marmsoler^{id} and Asad Ahmed and Achim D. Brucker^{id}

March 17, 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 Solidity's 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 | 7 |
| 2 | Preliminaries | 9 |
| 2.1 | Relators (Utils) | 9 |
| 2.2 | Fold (Utils) | 9 |
| 2.3 | Take (Utils) | 15 |
| 2.4 | Filter (Utils) | 16 |
| 2.5 | Those (Utils) | 16 |
| 2.6 | Fold Map (Utils) | 17 |
| 2.7 | Prefix (Utils) | 19 |
| 2.8 | Nth safe (Utils) | 20 |
| 2.9 | Some Basic Lemmas for Finite Maps (Utils) | 21 |
| 2.10 | Address class with example instantiation (Utils) | 21 |
| 2.11 | Common lemmas for sums (Utils) | 22 |
| 2.12 | Pred Some (Utils) | 23 |
| 2.13 | Termination Lemmas (Utils) | 24 |
| 2.14 | state Monad with Exceptions (State_Monad) | 25 |
| 3 | Stores and State | 35 |
| 3.1 | Memory (Memory) | 35 |
| 3.2 | Array Lookup (Memory) | 35 |
| 3.3 | Memory Lookup (Memory) | 38 |
| 3.4 | Memory Update (Memory) | 40 |
| 3.5 | Memory Locations (Memory) | 41 |
| 3.6 | Locations and Array Lookup (Memory) | 43 |
| 3.7 | Locations and Lookup (Memory) | 45 |
| 3.8 | Memory Locations (Memory) | 51 |
| 3.9 | Copy from Memory (Memory) | 63 |
| 3.10 | Copy Memory and Memory Locations (Memory) | 68 |
| 3.11 | Separation Check (Memory) | 70 |
| 3.12 | Array Data (Memory) | 86 |
| 3.13 | Array Lookup (Memory) | 86 |
| 3.14 | Array Lookup and Memory Copy (Memory) | 88 |
| 3.15 | Array Update (Memory) | 89 |
| 3.16 | Calldata Update and Memory Copy (Memory) | 93 |
| 3.17 | Initialize Memory (Memory) | 107 |
| 3.18 | Memory Init and Lookup (Memory) | 116 |
| 3.19 | Memory Init and Memory Locations (Memory) | 121 |
| 3.20 | Memory Init and Memory Copy (Memory) | 133 |
| 3.21 | Minit and Separation Check (Memory) | 135 |
| 3.22 | Calldata (Stores) | 138 |
| 3.23 | Storage (Stores) | 141 |
| 3.24 | Storage Lookup (Stores) | 141 |
| 3.25 | Storage Update (Stores) | 141 |
| 3.26 | Value types (State) | 142 |
| 3.27 | Common functions (State) | 143 |
| 3.28 | Operations on bytes (State) | 144 |
| 3.29 | State (State) | 145 |
| 4 | Expressions and Statements | 147 |
| 4.1 | Value types (Solidity) | 147 |
| 4.2 | Constants (Solidity) | 148 |

| | | |
|----------|---|------------|
| 4.3 | Unary Operations (Solidity) | 149 |
| 4.4 | Binary Operations (Solidity) | 150 |
| 4.5 | Store Lookups (Solidity) | 152 |
| 4.6 | Stack Lookups (Solidity) | 152 |
| 4.7 | Skip (Solidity) | 153 |
| 4.8 | Conditionals (Solidity) | 153 |
| 4.9 | Require/Assert (Solidity) | 154 |
| 4.10 | Stack Assign (Solidity) | 154 |
| 4.11 | Storage Assignment (Solidity) | 156 |
| 4.12 | Loops (Solidity) | 156 |
| 4.13 | Internal Method Calls (Solidity) | 157 |
| 4.14 | External Method Calls (Solidity) | 158 |
| 4.15 | Transfer (Solidity) | 158 |
| 4.16 | Solidity (Solidity) | 160 |
| 4.17 | Arrays (Solidity) | 160 |
| 4.18 | Declarations (Solidity) | 162 |
| 5 | ML Commands | 165 |
| 6 | Weakest Precondition Calculus | 167 |
| 6.1 | Weakest precondition calculus (WP) | 167 |
| 7 | Unit Tests | 183 |
| 7.1 | Examples (Unit_Tests) | 183 |
| 7.2 | States (Unit_Tests) | 184 |
| 7.3 | Constants (Unit_Tests) | 185 |
| 7.4 | Basic Operators (Unit_Tests) | 185 |
| 7.5 | Store lookup (Unit_Tests) | 187 |
| 7.6 | Stack lookup (Unit_Tests) | 188 |
| 7.7 | Conditionals (Unit_Tests) | 191 |
| 7.8 | Assignments (Unit_Tests) | 191 |
| 7.9 | Declarations (Unit_Tests) | 194 |
| 8 | Applications | 199 |
| 8.1 | Token Contract (Token) | 199 |
| 8.2 | Banking Contract (Bank) | 202 |
| 8.3 | Casino Contract (Casino) | 205 |
| 8.4 | Verifying an Invariant (Casino) | 208 |
| 8.5 | Voting Contract (Voting) | 212 |
| 8.6 | Simple Auction Contract (SimpleAuction) | 222 |
| 8.7 | Verifying an invariant (SimpleAuction) | 223 |
| 8.8 | Verifying an Invariant (SimpleAuction) | 225 |

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).

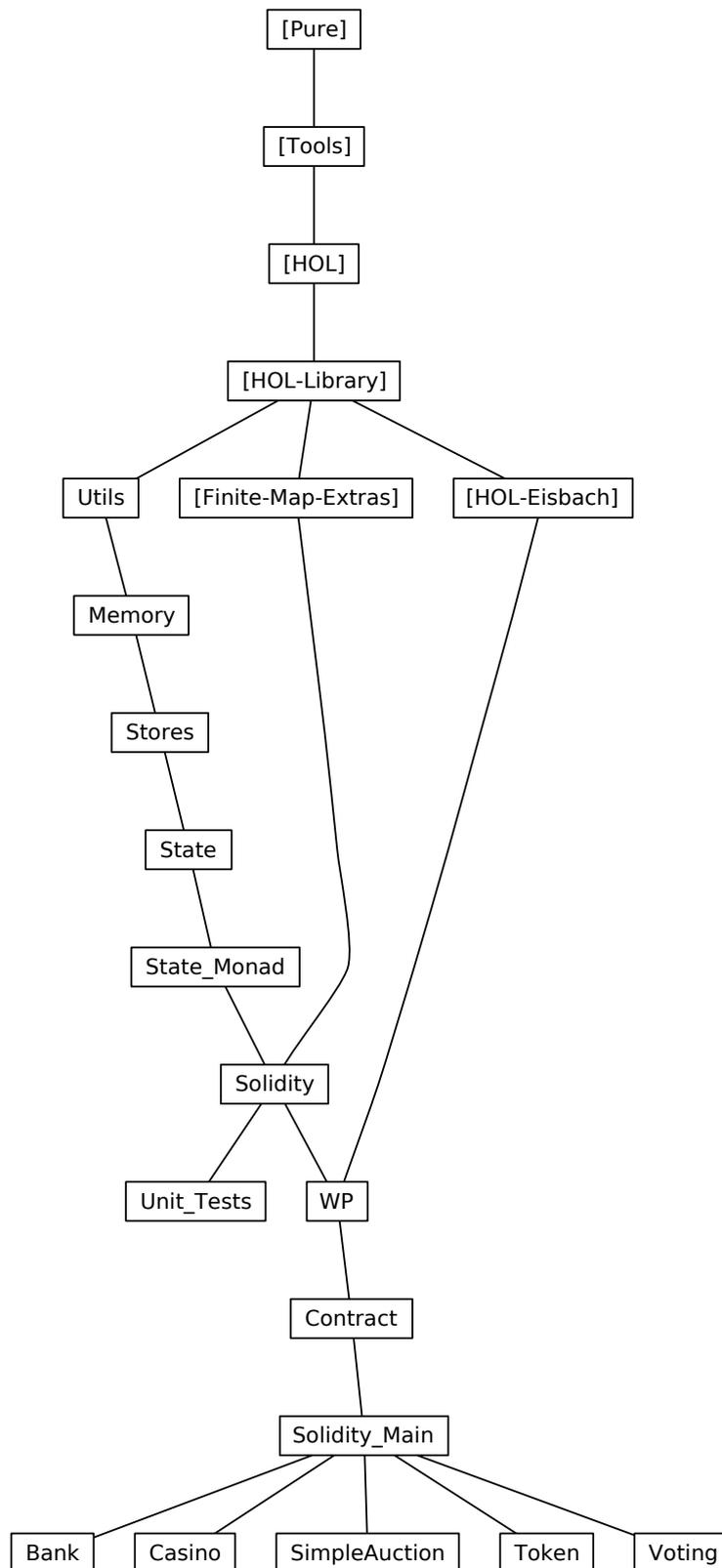


Figure 1.1: The Dependency Graph of the Isabelle Theories.

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 " $\bigwedge x. x \in \text{set } xs \implies (\bigwedge y. R \ x \ y = (f \ x = y))$ "
  shows " $\text{list\_all2 } R \ xs \ ys = (\text{map } f \ xs = ys)$ "
  using assms
proof (induction xs arbitrary: ys)
  case Nil
  then show ?case by blast
next
  case (Cons a xs)
  then show ?case
    by (smt (verit, best) list.set_intros(1,2) list.simps(9) list_all2_Cons1)
qed
```

2.2 Fold (Utils)

```
lemma fold_none_none[simp]:
  "fold ( $\lambda x \ y. y \gg= (\lambda y'. f \ x \ y')$ ) xs None = None"
  by (induction xs) (simp_all)

lemma fold_take:
  assumes " $n < \text{length } xs$ "
  shows " $\text{fold } f \ (\text{take } (\text{Suc } n) \ xs) \ s = f \ (xs!n) \ (\text{fold } f \ (\text{take } n \ xs) \ s)$ "
  using assms
proof (induction xs arbitrary: s n)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case by (cases n) (simp_all)
qed

lemma fold_some_take_some:
  assumes " $\text{fold } (\lambda x \ y. y \gg= (\lambda y'. f \ x \ y')) \ xs \ a = \text{Some } x$ "
  and " $n < \text{length } xs$ "
  obtains y where " $(\text{fold } (\lambda x \ y. y \gg= (\lambda y'. f \ x \ y')) \ (\text{take } n \ xs) \ a) \gg= (\lambda y'. f \ (xs!n) \ y') = \text{Some } y$ "
proof -
  have " $\forall n. n < \text{length } xs \longrightarrow (\exists y. (\lambda x \ y. y \gg= (\lambda y'. f \ x \ y')) \ (xs!n) \ (\text{fold } (\lambda x \ y. y \gg= (\lambda y'. f \ x \ y')) \ (\text{take } n \ xs) \ a) = \text{Some } y)$ "
  proof (rule ccontr)
    assume " $\neg (\forall n. n < \text{length } xs \longrightarrow (\exists y. \text{fold } (\lambda x \ y. y \gg= f \ x) \ (\text{take } n \ xs) \ a \gg= f \ (xs ! n) = \text{Some } y))$ "
    then obtain n where *: " $n < \text{length } xs$ " and **: " $\text{fold } (\lambda x \ y. y \gg= f \ x) \ (\text{take } n \ xs) \ a \gg= f \ (xs ! n) = \text{None}$ " by blast
    {
      fix n' assume " $n \leq n'$ "
```

```

then have "n' < length xs  $\longrightarrow$  fold ( $\lambda x y. y \ggg f x$ ) (take n' xs) a  $\ggg f$  (xs ! n') = None"
proof (induction rule: nat_induct_at_least)
  case base
  then show ?case using ** by simp
next
  case (Suc n')
  then show ?case by (simp add: fold_take)
qed
}
then have "fold ( $\lambda x y. y \ggg f x$ ) (take (length xs - 1) xs) a  $\ggg f$  (xs ! (length xs - 1)) = None"
  using "*" One_nat_def by auto
then have "fold ( $\lambda x y. y \ggg (\lambda y'. f x y')$ ) xs a = None"
  using * fold_take[of "(length xs - 1)" xs " $\lambda x y. y \ggg f x$ "] by auto
then show False using assms by simp
qed
then show ?thesis using that using assms(2) by blast
qed

```

lemma fold_same:

```

assumes " $\forall x \in \text{set } xs. f x = g x$ "
shows "fold ( $\lambda x y. y \ggg (\lambda y'. f x \ggg (\lambda l. \text{Some } (l \cup\! \cup y')$ ))) xs L
  = fold ( $\lambda x y. y \ggg (\lambda y'. g x \ggg (\lambda l. \text{Some } (l \cup\! \cup y')$ ))) xs L"
using assms
by (induction xs arbitrary: L, auto)

```

lemma fold_f_none_none[simp]:

```

assumes "f a = None"
shows "fold ( $\lambda x y. y \ggg (\lambda y'. f x \ggg (\lambda l. \text{Some } (l \cup\! \cup y')$ ))) (a # xs) (Some X) = None"
using assms by (induction xs) (simp_all)

```

lemma fold_none_the_fold:

```

assumes "(fold ( $\lambda x y. y \ggg (\lambda y'. f x \ggg (\lambda l. \text{Some } (l \cup\! \cup y')$ ))) xs (Some X))  $\neq$  None
   $\vee$  (fold ( $\lambda x y. y \ggg (\lambda y'. f x \ggg (\lambda l. \text{Some } (l \cup\! \cup y')$ ))) xs (Some (X  $\cup\! \cup$  Y)))  $\neq$  None"
shows "the (fold ( $\lambda x y. y \ggg (\lambda y'. f x \ggg (\lambda l. \text{Some } (l \cup\! \cup y')$ ))) xs (Some (X  $\cup\! \cup$  Y))) =
  (the (fold ( $\lambda x y. y \ggg (\lambda y'. f x \ggg (\lambda l. \text{Some } (l \cup\! \cup y')$ ))) xs (Some X)))  $\cup\! \cup$  Y"
using assms
proof (induction xs arbitrary: X)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  show ?case
  proof (cases "f a")
    case None
    then show ?thesis using Cons by auto
  next
    case (Some a')
    moreover from Cons(2) have
      "fold ( $\lambda x y. y \ggg (\lambda y'. f x \ggg (\lambda l. \text{Some } (l \cup\! \cup y')$ ))) xs (Some (a'  $\cup\! \cup$  X))  $\neq$  None
       $\vee$  fold ( $\lambda x y. y \ggg (\lambda y'. f x \ggg (\lambda l. \text{Some } (l \cup\! \cup y')$ ))) (a # xs) (Some (X  $\cup\! \cup$  Y))  $\neq$  None"
      using Some by (simp add: funion_assoc)
    ultimately show ?thesis
      using Cons(1)[of "(a'  $\cup\! \cup$  X)"] by (simp add: sup_assoc)
  qed
qed

```

lemma fold_some_some:

```

shows "fold ( $\lambda x y. y \ggg (\lambda y'. f x \ggg (\lambda l. \text{Some } (l \cup\! \cup y')$ ))) (take (n) xs) (Some (finsert x X))
  = fold ( $\lambda x y. y \ggg (\lambda y'. f x \ggg (\lambda l. \text{Some } (l \cup\! \cup y')$ ))) (take (n) xs) (Some X)  $\ggg$  Some  $\circ$ 
  finsert x"
proof (induction xs arbitrary: n X)
  case Nil
  then show ?case by simp
next

```

```

case (Cons a xs)
show ?case
proof (cases n)
  case 0
  then show ?thesis by auto
next
case (Suc n')
show ?thesis
proof (cases "f a")
  case None
  then show ?thesis using Suc by auto
next
case (Some a')
  then show ?thesis
  using Cons(1)[of n' "the (f a) |U| X"] Suc by auto
qed
qed
qed

```

lemma fold_insert_same:

```

assumes "x ∉ fset (the (fold (λx y. y ≫ (λy'. f x ≫ (λl. Some (l |U| y')))) (take n xs) (Some X)))"
shows "the (fold (λx y. y ≫ (λy'. f x ≫ (λl. Some (l |U| y')))) (take n xs) (Some X)) =
  (the (fold (λx y. y ≫ (λy'. f x ≫ (λl. Some (l |U| y')))) (take n xs) (Some (finsert x X)))) |-| {x}"
using assms
proof (induction xs arbitrary: n X)
  case Nil
  then show ?case by simp
next
case (Cons a xs)
show ?case
proof (cases n)
  case 0
  then show ?thesis using Cons by auto
next
case (Suc n')
show ?thesis
proof (cases "f a")
  case None
  then show ?thesis using Suc Cons by auto
next
case (Some a')
  then show ?thesis
  using Cons(1)[of n' "the (f a) |U| X"] Cons Suc by auto
qed
qed
qed

```

lemma fold_some_diff:

```

assumes "x ∉ fset (the (fold (λx y. y ≫ (λy'. f x ≫ (λl. Some (l |U| y')))) (take n xs) (Some {||})))"
and "(fold (λx y. y ≫ (λy'. f x ≫ (λl. Some (l |U| y')))) (take n xs) (Some {||})) ≠ None"
shows "fold (λx y. y ≫ (λy'. f x ≫ (λl. Some (l |U| y')))) (take n xs) (Some {||}) =
  Some ((the (fold (λx y. y ≫ (λy'. f x ≫ (λl. Some (l |U| y')))) (take n xs) (Some {x|})))
|-| {x}"
using fold_insert_same[OF assms(1)] assms by fastforce

```

lemma fold_none[simp]:

```

"fold (λx y. y ≫ g x) xs None = None"
proof (induction xs)
  case Nil
  then show ?case by simp
next

```

2 Preliminaries

```

case (Cons a xs)
then show ?case by auto
qed

```

lemma fold_some:

```

assumes "fold (λx y. y ≫ (λy'. (f x) ≫ (λl. Some (l |U| y')))) xs X0 = Some X"
shows "∃X'. X0 = Some X' ∧ X' |⊆| X"
using assms
proof (induction xs arbitrary: X0)
case Nil
then show ?case by simp
next
case (Cons a xs)
from Cons(2) obtain X' where "X0 = Some X'" by (case_tac X0, auto)
with Cons(2) have *: "(fold (λx y. y ≫ (λy'. f x ≫ (λl. Some (l |U| y')))) xs (f a ≫ (λl. Some (l |U| X')))) = Some X" by auto
with Cons(1) obtain X'' where X''_def: "f a ≫ (λl. Some (l |U| X')) = Some X''" and "X'' |⊆| X"
by auto
moreover from * X''_def have "fold (λx y. y ≫ (λy'. f x ≫ (λl. Some (l |U| y')))) xs (Some X'') = Some X"
by auto
ultimately show ?case using 'X0 = Some X'' by (case_tac "f a", auto)
qed

```

lemma fold_some_ex:

```

assumes "fold (λx y. y ≫ (λy'. f x ≫ (λl. Some (l |U| y')))) xs (Some X') = Some X"
and "i |∈| X"
and "i |∉| X'"
shows "∃x A. x ∈ set xs ∧ f x = Some A ∧ i |∈| A"
using assms
proof (induction xs arbitrary: X')
case Nil
then show ?case by simp
next
case (Cons a xs)
from Cons(2) obtain X'' where fa_def: "f a = Some X''" by fastforce
show ?case
proof (cases "i |∈| X''")
case True
then show ?thesis using Cons fa_def by auto
next
case False
moreover from Cons(2) have "fold (λx y. y ≫ (λy'. f x ≫ (λl. Some (l |U| y')))) xs (f a ≫ (λl. Some (l |U| X')))) = Some X" by auto
ultimately show ?thesis using Cons(1)[of "X'' |U| X'", OF _ Cons(3)] Cons(4) fa_def by auto
qed
qed

```

lemma fold_some_subs:

```

assumes "fold (λx y. y ≫ (λy'. f x ≫ (λl. Some (l |U| y')))) xs X' = Some X"
and "i ∈ set xs"
shows "∃A. f i = Some A ∧ A |⊆| X"
using assms
proof (induction xs arbitrary: X')
case Nil
then show ?case by simp
next
case (Cons a xs)
from Cons(2) obtain X'' where "X' = Some X''" by (case_tac X', auto)
with Cons have xx: "(fold (λx y. y ≫ (λy'. f x ≫ (λl. Some (l |U| y')))) xs (f a ≫ (λl. Some (l |U| X'')))) = Some X" by auto

then show ?case
proof (cases "i = a")

```

```

    case True
    with xx obtain XX where "(f i ≧ (λl. Some (l |U| X''))) = Some XX" and "XX |⊆| X" using
fold_some by blast
    then obtain XXX where "f i = Some XXX" and "XX = XXX |U| X'" by (cases "f i", auto)
    then show ?thesis using 'XX |⊆| X' by simp
  next
    case False
    then show ?thesis using xx Cons.IH Cons.prems(2) by auto
qed
qed

lemma fold_subs_none:
  assumes "fold (λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) xs (Some X) = None"
    and "X |⊆| Y"
  shows "fold (λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) xs (Some Y) = None"
  using assms
proof (induction xs arbitrary: X Y)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  then show ?case
  proof (cases "f x")
    case None
    then show ?thesis using Cons by simp
  next
    case (Some x')
    then have "fold (λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) (x # xs) (Some Y) =
      fold (λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) xs (Some (x' |U| Y))" by simp
    moreover have "fold (λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) xs (Some (x' |U| Y)) =
None"
    apply (rule Cons(1)) using Cons(2,3) Some by auto
    ultimately show ?thesis by auto
  qed
qed
qed

lemma fold_f_set_none:
  assumes "a ∈ set xs"
    and "f a = None"
  shows "fold (λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) xs (Some X) = None"
  using assms
proof (induction xs)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  then show ?case
  proof (cases "x=a")
    case True
    then show ?thesis using Cons by simp
  next
    case False
    then have "a ∈ set xs" using Cons by simp
    then have "fold (λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) xs (Some X) = None"
      using Cons by simp
    then show ?thesis by (cases "f x", auto simp add: fold_subs_none)
  qed
qed
qed

lemma fold_f_set_some:
  assumes "∀a ∈ set xs. f a ≠ None"
  shows "fold (λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) xs (Some X) ≠ None"
  using assms
proof (induction xs arbitrary: X)

```

```

  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  then obtain y where "(f x  $\gg$  ( $\lambda$ l. Some (1 | $\cup$ | X))) = Some (y | $\cup$ | X)" by auto
  moreover from Cons have "fold ( $\lambda$ x y. y  $\gg$  ( $\lambda$ y'. f x  $\gg$  ( $\lambda$ l. Some (1 | $\cup$ | y'))))) xs (Some (y | $\cup$ | X))  $\neq$  None" by simp
  ultimately show ?case by simp
qed

```

lemma fold_disj:

```

  assumes " $\forall$ x  $\in$  set xs.  $\forall$ L. f x = Some L  $\longrightarrow$  s | $\cap$ | L = {}"
    and "fold ( $\lambda$ x y. y  $\gg$  ( $\lambda$ y'. f x  $\gg$  ( $\lambda$ l. Some (1 | $\cup$ | y'))))) xs (Some X) = Some L"
    and "X | $\cap$ | s = {}"
  shows "s | $\cap$ | L = {}"
  using assms
proof (induction xs arbitrary:X)
  case Nil
  then show ?case by auto
next
  case (Cons a xs)
  from Cons(2) have " $\forall$ x $\in$ set xs.  $\forall$ L. f x = Some L  $\longrightarrow$  s | $\cap$ | L = {}" by simp
  moreover from Cons(3) obtain L'
    where *: "f a = Some L'"
    and "fold ( $\lambda$ x y. y  $\gg$  ( $\lambda$ y'. f x  $\gg$  ( $\lambda$ l. Some (1 | $\cup$ | y'))))) xs (Some (L' | $\cup$ | X)) = Some L"
  by (cases "f a"; simp)
  moreover have "L' | $\cap$ | s = {}" using Cons(2) * by auto
  ultimately show ?case using Cons(1,4) by blast
qed

```

lemma fold_union_in:

```

  assumes "fold ( $\lambda$ x y. y  $\gg$  ( $\lambda$ y'. f x  $\gg$  ( $\lambda$ l. Some (1 | $\cup$ | y'))))) xs (Some L) = Some L'"
    and "x | $\in$ | L'"
  shows "x | $\in$ | L  $\vee$  ( $\exists$ n L''. n < length xs  $\wedge$  f (xs ! n) = Some L''  $\wedge$  x | $\in$ | L'')"
  using assms
proof (induction xs arbitrary: L)
  case Nil
  then show ?case by auto
next
  case (Cons a xs)
  from Cons(2) have
    *: "fold
      ( $\lambda$ x y. y  $\gg$  ( $\lambda$ y'. f x  $\gg$  ( $\lambda$ l. Some (1 | $\cup$ | y')))))
      xs
      (f a  $\gg$  ( $\lambda$ l. Some (1 | $\cup$ | L)))
    = Some L'"
  by simp
  moreover from * obtain L''' where "f a = Some L'''" by fastforce
  ultimately have
    "fold ( $\lambda$ x y. y  $\gg$  ( $\lambda$ y'. f x  $\gg$  ( $\lambda$ l. Some (1 | $\cup$ | y'))))) xs (Some (L''' | $\cup$ | L)) = Some L'"
  by simp
  then have "x | $\in$ | (L''' | $\cup$ | L)  $\vee$  ( $\exists$ n L''. n < length xs  $\wedge$  f (xs ! n) = Some L''  $\wedge$  x | $\in$ | L'')"
  using Cons by blast
  then show ?case
proof
  assume "x | $\in$ | L''' | $\cup$ | L"
  then show ?thesis
proof
  assume "x | $\in$ | L'''"
  then show "x | $\in$ | L  $\vee$  ( $\exists$ n L''. n < length (a # xs)  $\wedge$  f ((a # xs) ! n) = Some L''  $\wedge$  x | $\in$ | L'')"
  using <f a = Some L''''> by auto
next
  assume "x | $\in$ | L"
  then show ?thesis by simp

```

```

    qed
  next
    assume "∃ n L''. n < length xs ∧ f (xs ! n) = Some L'' ∧ x |∈| L''"
    then show ?thesis by auto
  qed
qed

```

lemma fold_subs:

```

  assumes "∀ x ∈ set xs. ∀ L. f x = Some L → fset L ⊆ Y"
    and "fold (λ x y. y ≫ (λ y'. f x ≫ (λ l. Some (l |∪| y')))) xs (Some X) = Some L"
    and "fset X ⊆ Y"
  shows "fset L ⊆ Y"
  using assms
proof (induction xs arbitrary: X)
  case Nil
  then show ?case by auto
next
  case (Cons a xs)
  from Cons(2) have "∀ x ∈ set xs. ∀ L. f x = Some L → fset L ⊆ Y" by simp
  moreover from Cons(3) obtain L'
    where *: "f a = Some L'"
    and "fold (λ x y. y ≫ (λ y'. f x ≫ (λ l. Some (l |∪| y')))) xs (Some (L' |∪| X)) = Some L"
    by (cases "f a"; simp)
  moreover have "fset L' ⊆ Y" using Cons(2) * by auto
  ultimately show ?case using Cons(1,4) by blast
qed

```

lemma fold_in_subs:

```

  assumes "fold (λ x y. y ≫ (λ y'. f x ≫ (λ l. Some (l |∪| y')))) xs (Some X) = Some L"
    and "x ∈ set xs"
    and "f x = Some S"
  shows "S |⊆| L"
  using assms
proof (induction xs arbitrary: X)
  case Nil
  then show ?case by auto
next
  case (Cons a xs)
  from Cons(2) obtain L'
    where *: "f a = Some L'"
    and **: "fold (λ x y. y ≫ (λ y'. f x ≫ (λ l. Some (l |∪| y')))) xs (Some (L' |∪| X)) = Some L"
    by (cases "f a"; simp)
  then have "L' |⊆| L" using fold_some[of f xs "(Some (L' |∪| X))" L] by simp
  show ?case
  proof (cases "a = x")
    case True
    then show ?thesis using <L' |⊆| L>
      using "*" assms(3) by auto
  next
    case False
    then show ?thesis using Cons(1)[OF **] Cons(3)
      using assms(3) by auto
  qed
qed

```

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"
  using assms
  by auto

```

```

lemma take_all1:
  assumes " $\forall n \leq \text{length } xs. P (\text{take } n \text{ } xs)$ "
  shows "P xs"
  using assms
  by auto

lemma rev_induct2 [consumes 1, case_names Nil snoc]:
  assumes "length xs = length ys" and "P [] []"
  and " $(\bigwedge x \text{ } xs \text{ } y \text{ } ys. \text{length } xs = \text{length } ys \implies P \text{ } xs \text{ } ys \implies P (xs @ [x]) (ys @ [y]))$ "
  shows "P xs ys"
proof -
  have "P (rev (rev xs)) (rev (rev ys))"
  by (rule_tac xs = "rev xs" and ys = "rev ys" in list_induct2, simp_all add: assms)
  then show ?thesis by simp
qed

```

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)))"
  using assms
proof (induction daa arbitrary: n)
  case Nil
  then show ?case by simp
next
  case (Cons a x)
  then show ?case
  proof (cases n)
    case 0
    then show ?thesis
    using Cons.prem1(2) by auto
  next
    case (Suc nat)
    then have "nat < length (x)"
    using Cons.prem1(1) by auto
    moreover have "P (x ! nat)"
    using Cons.prem1(2) Suc by auto
    ultimately have "length (filter P (take (Suc nat) x)) = Suc (length (filter P (take nat x)))"
    using Cons.IH by blast
    then show ?thesis by (simp add: Suc)
  qed
qed

```

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  $\wedge$  f x = Some y"
  using assms
  by (induction xs ys rule: list_induct2; simp split:option.split_asm)

lemma those_map_eq:
  assumes " $\forall x \in \text{set } xs. \forall y. f \text{ } x = \text{Some } y \implies g \text{ } x = \text{Some } y$ "
  and " $\forall x \in \text{set } xs. f \text{ } x \neq \text{None}$ "
  shows "those (map f xs) = those (map g xs)"
  using assms
  by (metis list.map_cong0 option.exhaust)

lemma those_map_none:
  assumes "those (map f xs) = Some y"
  shows " $\forall x \in \text{set } xs. f \text{ } x \neq \text{None}$ "
proof (rule ccontr)

```

```

assume "¬ (∀x∈set xs. f x ≠ None)"
then obtain x where "x∈set xs" and "f x = None" by auto
then have "those (map f xs) = None"
proof (induction xs)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case by (auto split:option.split)
qed
then show False using assms by simp
qed

lemma those_map_some[simp]:
  "those (map Some xs) = Some xs"
by (induction xs) (simp_all)

lemma those_some_map:
  assumes "those xs = Some xs'"
  shows "xs = map Some xs'"
using assms by (induction xs arbitrary:xs') (auto split:option.split_asm)

lemma those_those:
  assumes "those xs = Some xs'"
  and "those ys = Some ys'"
  shows "(those (xs @ ys)) = Some (xs' @ ys)'"
by (metis assms(1) assms(2) map_append those_map_some those_some_map)

lemma those_map_none_none:
  assumes "those (map f as) = None"
  shows "∃x ∈ set as. f x = None"
using assms
by (induction as;fastforce)

lemma those_map_some_some:
  assumes "∀x ∈ set as. f x ≠ None"
  shows "those (map f as) ≠ None"
using those_map_none_none assms by blast

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"
using assms
by (metis map_equality_iff nth_map those_map_some)

```

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 (induction ds arbitrary: m)
  case Nil
  then show ?case by simp
next
  case (Cons a ds)
  from Cons(1)[of "snd (f a m)"] show ?case by (auto split:prod.split)
qed

```

lemma fold_map_mono:

assumes " $\bigwedge x y x' y'. (f x y) = (x', y') \implies f' y' \geq ((f' y):: \text{nat})$ "
and "fold_map f x y = (x', y')"

shows " $f' y' \geq f' y$ "

using assms by (induction x arbitrary: y x' y', simp) (force split:prod.split prod.split_asm)

lemma fold_map_geq:

assumes " $\bigwedge y x. f' (\text{snd } (f y x)) \geq ((f' x):: \text{nat})$ "

shows " $f' (\text{snd } (\text{fold_map } f x y)) \geq f' y$ "

proof (rule fold_map_mono[of f f' x])

show "fold_map f x y = (fst (fold_map f x y), snd (fold_map f x y))" by simp

next

fix x y x' y'

assume "f x y = (x', y')"

then show "f' y ≤ f' y'" using assms by (metis snd_conv)

qed

lemma fold_map_cong [fundef_cong]:

"a = b \implies xs = ys \implies ($\bigwedge x. x \in \text{set } xs \implies f x = g x$)

$\implies \text{fold_map } f xs a = \text{fold_map } g ys b$ "

by (induct ys arbitrary: a b xs) simp_all

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)))"

using assms

proof (induction xs arbitrary: m n)

case Nil

then show ?case by simp

next

case (Cons a xs)

show ?case

proof (cases n)

case 0

then show ?thesis by (auto split:prod.split)

next

case (Suc n')

then have "n' < length (fst (fold_map f xs (snd (f a m))))" using Cons(2) by (auto

split:prod.split_asm)

then have "fst (fold_map f xs (snd (f a m))) ! n' = fst (f (xs ! n') (snd (fold_map f (take n' xs) (snd (f a m)))))" using Cons(1)[of "n'" "(snd (f a m))"] by simp

moreover have "fst (fold_map f (a # xs) m) ! Suc n' = fst ((fold_map f xs (snd (f a m))) ! n'"

by (simp split:prod.split)

moreover have "(a # xs) ! Suc n' = xs ! n'" by simp

moreover have "(snd (fold_map f (take n' xs) (snd (f a m)))) = (snd (fold_map f (take (Suc n') (a # xs) m))" by (simp split:prod.split)

ultimately show ?thesis by (simp add: Suc)

qed

qed

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)))"

using assms

proof (induction xs arbitrary: m n)

case Nil

then show ?case by simp

next

case (Cons a xs)

show ?case

proof (cases n)

case 0

then show ?thesis by (auto split:prod.split)

```

next
  case (Suc n')
  then have "snd (fold_map f (take (Suc n') xs) (snd (f a m)))
            = snd (f (xs ! n') (snd (fold_map f (take n' xs) (snd (f a m)))))" using Cons by auto
  then show ?thesis using Suc by (auto split:prod.split)
qed
qed

```

2.7 Prefix (Utils)

```

definition prefix where "prefix m0 mf = ( $\exists m'''. mf = m0@m'''$ )"

```

```

lemma prefix_id[intro]: "prefix x x" unfolding prefix_def by simp

```

```

lemma prefix_trans: "prefix x y  $\implies$  prefix y z  $\implies$  prefix x z" unfolding prefix_def by fastforce

```

```

definition sprefix where "sprefix m0 mf = ( $\exists m'''. mf = m0@m'''$ )"

```

```

lemma sprefix_append[simp]: "sprefix xs (xs@[y])" unfolding sprefix_def by blast

```

```

lemma sprefix_prefix: "sprefix m0 mf  $\implies$  prefix m0 mf" unfolding prefix_def sprefix_def by auto

```

```

lemma sprefix_trans: "sprefix x y  $\implies$  sprefix y z  $\implies$  sprefix x z" unfolding sprefix_def by fastforce

```

```

lemma sprefix_length: "sprefix x y  $\implies$  length x < length y" unfolding sprefix_def by auto

```

```

lemma fold_map_prefix:
  assumes "fold_map f ds m = (ls, m')"
  and " $\bigwedge x y x' y'. f x y = (x', y') \implies$  prefix y y'"
  shows "prefix m m'"
  using assms

```

```

proof (induction ds arbitrary: m m' ls)
  case Nil
  then show ?case unfolding prefix_def by simp

```

```

next
  case (Cons a ds)
  from Cons(2) show ?case
  proof (auto split:prod.split_asm)
    fix x1 x2 x1a
    assume *: "f a m = (x1, x2)"
    and "fold_map f ds x2 = (x1a, m'"
    and "ls = x1 # x1a"
    then have "prefix x2 m'" using Cons by simp
    moreover have "prefix m x2" using Cons(3)[OF *] by simp
    ultimately show "prefix m m'" unfolding prefix_def by auto
  qed
qed

```

```

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

```

```

primrec ofold :: "('a  $\implies$  'b  $\implies$  'b option)  $\implies$  'a list  $\implies$  'b  $\implies$  'b option" where
  fold_Nil: "ofold f [] = Some" |
  fold_Cons: "ofold f (x # xs) b = ofold f xs b  $\ggg$  f x"

```

```

lemma ofold_cong [fundef_cong]:
  "a = b  $\implies$  xs = ys  $\implies$  ( $\bigwedge x. x \in$  set xs  $\implies$  f x = g x)
   $\implies$  ofold f xs a = ofold g ys b"
  by (induct ys arbitrary: a b xs) simp_all

```

```

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 \equiv List.case_list a b l"

2.8 Nth safe (Utils)

definition nth_safe :: "'a list \Rightarrow nat \Rightarrow '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 \implies xs \$ i = Some (xs!i)" unfolding nth_safe_def by simp

lemma nth_safe_none[simp]: "i \geq length xs \implies xs \$ i = None" unfolding nth_safe_def by simp

lemma nth_safe_length: "xs \$ i = Some x \implies i < length xs" unfolding nth_safe_def by (simp split: if_split_asm)

definition list_update_safe :: "'a list \Rightarrow nat \Rightarrow 'a \Rightarrow ('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 \ggg f = xs \$ x"

proof -

from assms(1) have "length as = length xs" using those_some_map map_equality_iff by blast
 then show ?thesis using assms

proof (induction as xs arbitrary: x rule: list_induct2)

case Nil

then show ?case by simp

next

case (Cons x' xs' y' ys')

then have *: "those (map f xs') = Some ys'" and **: "f x' = Some y'" using those_map by metis+

from * have ***: " $\bigwedge x. xs' $ x \ggg f = ys' $ x$ " using Cons(2) by blast

then have "(x' # xs') \$ x \ggg f = (y' # ys') \$ x"

proof (cases x)

case 0

then show ?thesis

using ** by (auto)

next

case (Suc nat)

moreover from *** have "(xs') \$ nat \ggg f = (ys') \$ nat" by simp

ultimately show ?thesis

by (metis Suc_less_eq length_Cons nth_Cons_Suc nth_safe_def)

qed

then show ?case by auto

qed

qed

lemma nth_in_set:
 assumes "xs \$ x = Some y"
 shows "y \in set xs"
 using assms unfolding nth_safe_def by (auto split:if_split_asm)

lemma set_nth_some:
 assumes "y \in set xs"
 shows " $\exists x. xs $ x = Some y$ "
 using assms unfolding nth_safe_def
 by (metis in_set_conv_nth)

lemma those_map_some_nth:
 assumes "those (map f as) = Some a"
 and "as \$ x = Some y"
 obtains z where "f y = Some z"
 using assms
 by (metis not_None_eq nth_mem nth_safe_def option.inject those_map_none)

lemma nth_safe_prefix:
 assumes "m \$ l = Some v"

```

and "prefix m m'"
shows "m' $ l = Some v"
  using assms unfolding prefix_def nth_safe_def
  by (auto split:if_split_asm simp add: nth_append_left)

```

2.9 Some Basic Lemmas for Finite Maps (Utils)

```

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

```

```

lemma fmlookup_finite:
  fixes f :: "'a  $\Rightarrow$  'a"
  and y :: "('a, 'b) fmap"
  assumes "inj_on ( $\lambda$ (ad, x). (f ad, x)) {(ad, x). (fmlookup y  $\circ$  f) ad = Some x}"
  shows "finite {(ad, x). (fmlookup y  $\circ$  f) ad = Some x}"
proof (cases rule: inj_on_finite[OF assms(1), of "{(ad, x)|ad x. (fmlookup y) ad = Some x}"])
  case 1
  then show ?case by auto
next
  case 2
  then have *: "finite {(ad, x) |ad x. fmlookup y ad = Some x}" using fmfinite[of y] by simp
  show ?case using finite_image_set[OF *, of " $\lambda$ (ad, x). (ad, x)"] by simp
qed

```

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" unfolding aspace_carrier_def by simp

```

```

typedef aspace = aspace_carrier
  unfolding aspace_carrier_def
  by (rule_tac x = 0 in exI) simp

```

```

setup_lifting type_definition_aspace

```

```

lift_definition A0 :: aspace is 0 unfolding aspace_carrier_def by simp
lift_definition A1 :: aspace is 1 unfolding aspace_carrier_def by simp
lift_definition A2 :: aspace is 2 unfolding aspace_carrier_def by simp
lift_definition A3 :: aspace is 3 unfolding aspace_carrier_def by simp
lift_definition A4 :: aspace is 4 unfolding aspace_carrier_def by simp
lift_definition A5 :: aspace is 5 unfolding aspace_carrier_def by simp
lift_definition A6 :: aspace is 6 unfolding aspace_carrier_def by simp
lift_definition A7 :: aspace is 7 unfolding aspace_carrier_def by simp
lift_definition A8 :: aspace is 8 unfolding aspace_carrier_def by simp
lift_definition A9 :: aspace is 9 unfolding aspace_carrier_def by simp

```

```

lemma aspace_finite: "finite (UNIV::aspace set)"
  using finite_imageI[OF aspace_carrier_finite, of Abs_aspace]
  type_definition.Abs_image[OF type_definition_aspace] by simp

```

```

lemma A1_neq_A0[simp]: "A1  $\neq$  A0"

```

```

by transfer simp

instantiation aspace :: address
begin
  definition null_def: "null = A0"
  instance proof
    show "finite (UNIV::aspace set)" using aspace_finite by simp
  qed
end

instantiation aspace :: equal
begin

definition "HOL.equal (x::aspace) y  $\longleftrightarrow$  Rep_aspace x = Rep_aspace y"

instance apply standard by (simp add: Rep_aspace_inject equal_aspace_def)

end

```

2.11 Common lemmas for sums (Utils)

```

lemma sum_addr:
  fixes f::('a::address) $\Rightarrow$ nat"
  shows "( $\sum$  ad $\in$ UNIV. f ad) = ( $\sum$  ad/ad  $\neq$  addr. f ad) + f addr"
proof -
  have "finite {ad  $\in$  UNIV. ad  $\neq$  addr}" and "finite {ad  $\in$  UNIV. ad = addr}" using finite by simp+
  moreover have "UNIV = {ad  $\in$  UNIV. ad  $\neq$  addr}  $\cup$  {ad  $\in$  UNIV. ad = addr}" by auto
  moreover have "{ad  $\in$  UNIV. ad  $\neq$  addr}  $\cap$  {ad  $\in$  UNIV. ad = addr} = {}" by auto
  ultimately have "sum f UNIV = sum f {ad  $\in$  UNIV. ad  $\neq$  addr} + sum f {ad  $\in$  UNIV. ad = addr}"
    using sum_Un_nat[of "{ad $\in$ UNIV. ad  $\neq$  addr}" "{ad $\in$ UNIV. ad = addr}" f] by simp
  moreover have "sum f {ad  $\in$  UNIV. ad = addr} = f addr" by simp
  ultimately show ?thesis by simp
qed

lemma sum_addr2:
  fixes f::"'a::address $\Rightarrow$ nat"
  assumes "addr  $\in$  A"
  shows "( $\sum$  ad $\in$ A. f ad) = ( $\sum$  ad/ad $\in$ A  $\wedge$  ad  $\neq$  addr. f ad) + f addr"
proof -
  have "finite {ad  $\in$  A. ad  $\neq$  addr}" and "finite {ad  $\in$  A. ad = addr}" using finite by simp+
  moreover have "A = {ad  $\in$  A. ad  $\neq$  addr}  $\cup$  {ad  $\in$  A. ad = addr}" by auto
  moreover have "{ad  $\in$  A. ad  $\neq$  addr}  $\cap$  {ad  $\in$  A. ad = addr} = {}" by auto
  ultimately have "( $\sum$  ad $\in$ A. f ad) = ( $\sum$  ad/ad $\in$ A  $\wedge$  ad  $\neq$  addr. f ad) + ( $\sum$  ad/ad $\in$ A  $\wedge$  ad = addr. f ad)"
    using sum_Un_nat[of "{ad $\in$ A. ad  $\neq$  addr}" "{ad $\in$ A. ad = addr}" f] by simp
  moreover have "{ad  $\in$  A. ad = addr} = {addr}" using assms by auto
  then have "sum f {ad  $\in$  A. ad = addr} = f addr" by simp
  ultimately show ?thesis by simp
qed

lemma sum_addr3:
  fixes f::"'a::address $\Rightarrow$ nat"
  assumes "addr  $\notin$  A"
  shows "( $\sum$  ad $\in$ A. f ad) = ( $\sum$  ad/ad $\in$ A  $\wedge$  ad  $\neq$  addr. f ad)"
proof -
  have "finite {ad  $\in$  A. ad  $\neq$  addr}" and "finite {ad  $\in$  A. ad = addr}" using finite by simp+
  moreover have "A = {ad  $\in$  A. ad  $\neq$  addr}  $\cup$  {ad  $\in$  A. ad = addr}" by auto
  moreover have "{ad  $\in$  A. ad  $\neq$  addr}  $\cap$  {ad  $\in$  A. ad = addr} = {}" by auto
  ultimately have "( $\sum$  ad $\in$ A. f ad) = ( $\sum$  ad/ad $\in$ A  $\wedge$  ad  $\neq$  addr. f ad) + ( $\sum$  ad/ad $\in$ A  $\wedge$  ad = addr. f ad)"
    using sum_Un_nat[of "{ad $\in$ A. ad  $\neq$  addr}" "{ad $\in$ A. ad = addr}" f] by simp
  moreover have "{ad  $\in$  A. ad = addr} = {}" using assms by simp
  then have "sum f {ad  $\in$  A. ad = addr} = 0" by simp
  ultimately show ?thesis by simp

```

qed

2.12 Pred Some (Utils)

```

definition pred_some where
  "pred_some P v = ( $\exists v'$ . v = Some v'  $\wedge$  P v')"
```

```

definition fs_disj_fs where
  "fs_disj_fs B C = pred_some ( $\lambda C'$ . pred_some ( $\lambda B'$ . C' | $\cap$ | B' = {||}) B) C"
```

```

definition s_disj_fs where
  "s_disj_fs B C = pred_some ( $\lambda C'$ . fset C'  $\cap$  B = {}) C"
```

```

definition s_eq_fs where
  "s_eq_fs B C = pred_some ( $\lambda C'$ . B = fset C') C"
```

```

definition s_subs_fs where
  "s_subs_fs B C = pred_some ( $\lambda C'$ . B  $\subseteq$  fset C') C"
```

```

definition fs_subs_fs where
  "fs_subs_fs B C = pred_some ( $\lambda B'$ . B' | $\subseteq$ | C) B"
```

```

definition fs_subs_s where
  "fs_subs_s B C = pred_some ( $\lambda B'$ . fset B'  $\subseteq$  C) B"
```

```

definition s_union_fs where
  "s_union_fs A B C = pred_some ( $\lambda C'$ . A = B  $\cup$  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"
  using assms unfolding prefix_def s_disj_fs_def pred_some_def loc_def by fastforce
```

```

lemma s_union_fs_s_union_fs_union:
  assumes "s_union_fs B X B'"
    and "A = (B - C)  $\cup$  D"
    and "A' = Some ((the B' | $-$ | C') | $\cup$ | the D')"
    and "C  $\cap$  X = {}"
    and "C = fset C'"
    and "D = fset (the D')"
  shows "s_union_fs A X A'"
  using assms unfolding s_union_fs_def pred_some_def
  by fastforce
```

```

lemma s_union_fs_diff:
  assumes "s_union_fs A B C"
    and "B  $\cap$  fset (the C) = {}"
  shows "(A - B) = fset (the C)"
  using assms unfolding s_union_fs_def pred_some_def by auto
```

```

lemma s_disj_fs_loc_fold:
  assumes "s_disj_fs (loc m0) (fold ( $\lambda x y. y \gg= (\lambda y'. f x \gg= (\lambda l. Some (l | $\cup$ | y'))$ )) xs (X))"
    and "s_disj_fs (loc m0) X"
  shows "s_disj_fs (loc m0) (fold ( $\lambda x y. y \gg= (\lambda y'. f x \gg= (\lambda l. Some (l | $\cup$ | y'))$ )) (take n xs) (X))"
  using assms
```

```

proof (induction xs arbitrary:n X)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)

```

```

then show ?case
proof (cases n)
  case 0
  then show ?thesis using Cons by simp
next
case (Suc nat)
show ?thesis
proof (cases "f a")
  case None
  then show ?thesis using Suc Cons by simp
next
case (Some a')
  moreover have "s_disj_fs (loc m0) (fold (λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) xs
((λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) a X))" using Cons(2) by auto
  moreover have "s_disj_fs (loc m0) ((λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) a X)"
  proof -
    obtain y where *: "X = Some y" using Cons(3) unfolding s_disj_fs_def pred_some_def by blast
    moreover from Cons(2) have "s_disj_fs (loc m0) (fold (λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) xs (Some (l |U| y')))) xs (Some (a' |U| y))"
    using Some * by simp
    moreover from Cons(2) have "fold (λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) xs (Some (l |U| y)) (a' |U| y)) ≠ None"
    using * Some unfolding s_disj_fs_def pred_some_def by auto
    then have "the (fold (λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) xs (Some (a' |U| y)))
= the (fold (λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) xs (Some {l})) |U| (a' |U| y)"
    using fold_none_the_fold[of f xs "{l}" "(a' |U| y)"] by simp
    ultimately show ?thesis using Some unfolding s_disj_fs_def pred_some_def by auto
  qed
  ultimately show ?thesis
  using Cons(1)[of "((λx y. y ≧ (λy'. f x ≧ (λl. Some (l |U| y')))) a (X))" nat] Suc
  by auto
qed
qed
qed
qed

```

```

lemma s_disj_union_fs: "s_disj_fs B C ⇒ s_union_fs A B C ⇒ fset (the C) = A - B"
unfolding s_disj_fs_def s_union_fs_def pred_some_def by fastforce

```

```

lemma disj_empty[simp]: "s_disj_fs X (Some {l})" unfolding s_disj_fs_def pred_some_def by simp

```

```

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'"
  using assms unfolding s_union_fs_def pred_some_def
  by fastforce

```

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 -
  have "card ({0..length m} - insert p1 (fset s1)) = card ({0..length m}) - card ({0..length m} ∩
insert p1 (fset s1))"
  and "card ({0..length m} - fset s1) = card ({0..length m}) - card ({0..length m} ∩ (fset s1))" by
(rule card_Diff_subset_Int, simp)+
  moreover from assms(2) have "card ({0..length m} ∩ insert p1 (fset s1)) = Suc (card ({0..length m}
∩ (fset s1)))" using nth_safe_length[OF assms(1)] by simp
  moreover have "card ({0..length m}) ≥ card ({0..length m} ∩ insert p1 (fset s1))" by (rule

```

```

card_mono, auto)
  ultimately show ?thesis by simp
qed

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 (cases x)
  case (Normal n)
  then show ?thesis using n by force
next
  case (Exception e)
  then show ?thesis using e by force
next
  case NT
  then show ?thesis using t by simp
qed

typedef ('a, 'e, 's) state_monad = "UNIV::('s ⇒ ('a × 's, 'e × 's) result) set"
morphisms execute create
  by simp

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))"
  by (simp add: Let_def)

```

2.14.1 Code Generator Setup

```

code_datatype create

lemma execute_create[execute_simps, code]: "execute (create f) = f" using create_inverse by simp

declare execute_inverse[simp]

lemma execute_ext[intro]: "( $\bigwedge x. (execute\ m1\ x = execute\ m2\ x) \implies m1 = m2$ )" using HOL.ext
  by (metis execute_inverse)

```

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"
  unfolding return_def by (auto simp add:execute_simps)

lemma execute_returnE:
  assumes "execute (return x) s = Normal (a, s'"

```

```

shows "x = a" and "s = s'"
using assms unfolding return_def execute_create by auto

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)"
  unfolding throw_def by (auto simp add:execute_simps)

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"
  unfolding bind_def execute_create by simp_all

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'"
  using assms unfolding bind_def execute_create apply (cases "execute f s") using that by auto

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'"
  using assms unfolding bind_def execute_create apply (cases "execute f s") using that by auto

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)"
  unfolding bind_def
proof -
  have "(λs. case execute m1 s of Normal (a, xa) ⇒ execute (m3 a) xa | Exception x ⇒ Exception x | NT
⇒ NT) =
  (λs. case execute m2 s of Normal (a, xa) ⇒ execute (m4 a) xa | Exception x ⇒ Exception x | NT
⇒ NT)"
  (is "(λs. ?L s) = (λs. ?R s)")
  proof
    fix s
    show "?L s = ?R s"
    using assms by (cases "execute m1 s"; simp)
  qed
  then show "create (λs. ?L s) = create (λs. ?R s)" by simp
qed

lemma throw_left[simp]: "throw x >>= y = throw x" unfolding throw_def bind_def by (simp add:
execute_simps)

```

2.14.3 The Monad Laws

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

```
lemma return_bind [simp]: "(return x >>= f) = f x"
unfolding return_def bind_def by (simp add: execute_simps)
```

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

```
lemma bind_return [simp]: "(m >>= return) = m"
proof (rule execute_ext)
```

```
  fix s
  show "execute (m >>= return) s = execute m s"
  proof (cases "execute m s" rule: result_cases)
    case (n a s)
    then show ?thesis by (simp add: execute_simps)
  next
    case (e e)
    then show ?thesis by (simp add: execute_simps)
  next
    case t
    then show ?thesis by (simp add: execute_simps)
  qed
qed
```

($\gg=$) is associative

```
lemma bind_assoc:
  fixes m :: "('a,'e,'s) state_monad"
  fixes f :: "'a  $\Rightarrow$  ('b,'e,'s) state_monad"
  fixes g :: "'b  $\Rightarrow$  ('c,'e,'s) state_monad"
  shows "(m >>= f) >>= g = m >>= ( $\lambda$ x. f x >>= g)"
proof
  fix s
  show "execute (m >>= f >>= g) s = execute (m >>= ( $\lambda$ x. f x >>= g)) s"
  unfolding bind_def by (cases "execute m s" rule: result_cases; simp add: execute_simps)
qed
```

2.14.4 Basic Congruence Rules

```
lemma bind_case_nat_cong [fundef_cong]:
  assumes "x = x'" and " $\bigwedge$ a. x = Suc a  $\Longrightarrow$  f a h = f' a h"
  shows "(case x of Suc a  $\Rightarrow$  f a | 0  $\Rightarrow$  g) h = (case x' of Suc a  $\Rightarrow$  f' a | 0  $\Rightarrow$  g) h"
  by (metis assms(1) assms(2) old.nat.exhaust old.nat.simps(4) old.nat.simps(5))
```

```
lemma if_cong[fundef_cong]:
  assumes "b = b'"
  and "b'  $\Longrightarrow$  m1 s = m1' s"
  and " $\neg$  b'  $\Longrightarrow$  m2 s = m2' s"
  shows "(if b then m1 else m2) s = (if b' then m1' else m2') s"
  using assms(1) assms(2) assms(3) by auto
```

```
lemma bind_case_pair_cong [fundef_cong]:
  assumes "x = x'" and " $\bigwedge$ a b. x = (a,b)  $\Longrightarrow$  f a b s = f' a b s"
  shows "(case x of (a,b)  $\Rightarrow$  f a b) s = (case x' of (a,b)  $\Rightarrow$  f' a b) s"
  by (simp add: assms(1) assms(2) prod.case_eq_if)
```

```
lemma bind_case_let_cong [fundef_cong]:
  assumes "M = N"
  and " $\bigwedge$ x. x = N  $\Longrightarrow$  f x s = g x s"
  shows "(Let M f) s = (Let N g) s"
  by (simp add: assms(1) assms(2))
```

```
lemma bind_case_some_cong [fundef_cong]:
  assumes "x = x'" and " $\bigwedge$ a. x = Some a  $\Longrightarrow$  f a s = f' a s" and "x = None  $\Longrightarrow$  g s = g' s"
  shows "(case x of Some a  $\Rightarrow$  f a | None  $\Rightarrow$  g) s = (case x' of Some a  $\Rightarrow$  f' a | None  $\Rightarrow$  g') s"
  by (simp add: assms(1) assms(2) assms(3) option.case_eq_if)
```

```
lemma bind_case_bool_cong [fundef_cong]:
  assumes "x = x'" and "x = True  $\Longrightarrow$  f s = f' s" and "x = False  $\Longrightarrow$  g s = g' s"
```

```
shows "(case x of True ⇒ f | False ⇒ g) s = (case x' of True ⇒ f' | False ⇒ g') s"
using assms(1) assms(2) assms(3) by auto
```

2.14.5 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))"
  unfolding get_def by (auto simp add:execute_simps)
```

```
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))"
  unfolding put_def by (auto simp add:execute_simps)
```

```
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))"
  unfolding update_def by (auto simp add:execute_simps)
```

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)"
  unfolding modify_def by (auto simp add:execute_simps)
```

```
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)))" ..
```

```
lemma "do {
  x ← return 1;
  return 2;
  return x
}
```

```

    } = return 1"
  by auto

```

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"
unfolding option_def by (auto simp add:execute_simps)

```

```

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"
unfolding assert_def by (auto simp add:execute_simps)

```

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))"
  using assms unfolding effect_def by auto

```

```

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"
  by (simp add: sm_lub_def img_lub_def fun_lub_def flat_lub_def)

```

```

lemma sm_ordI:
  assumes "∧h. execute x h = NT ∨ execute x h = execute y h"

```

2 Preliminaries

```
shows "sm_ord x y"
using assms unfolding sm_ord_def img_ord_def fun_ord_def flat_ord_def
by blast
```

```
lemma sm_ordE:
  assumes "sm_ord x y"
  obtains "execute x h = NT" | "execute x h = execute y h"
  using assms unfolding sm_ord_def img_ord_def fun_ord_def flat_ord_def
  by atomize_elim blast
```

```
lemma sm_interpretation: "partial_function_definitions sm_ord sm_lub"
proof -
  have "partial_function_definitions (fun_ord result_ord) (fun_lub result_lub)"
    by (rule partial_function_lift) (rule flat_interpretation)
  then have "partial_function_definitions (img_ord execute (fun_ord result_ord))
    (img_lub execute create (fun_lub result_lub))"
    by (rule partial_function_image) (auto simp add: execute_simps)
  then show "partial_function_definitions sm_ord sm_lub"
    by (simp only: sm_ord_def sm_lub_def)
qed
```

```
interpretation sm: partial_function_definitions sm_ord sm_lub
  rewrites "sm_lub {}  $\equiv$  empty_result"
by (fact sm_interpretation) (simp add: sm_lub_empty)
```

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  $\Rightarrow$  's  $\Rightarrow$  's  $\Rightarrow$  'a  $\Rightarrow$  bool" where
  success_def: "success sm s s' a  $\longleftrightarrow$  execute sm s  $\neq$  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) ( $\lambda xa. \forall h r. is\_Normal (xa h) \wedge r = Inl$ 
  (normal (xa h))  $\vee$  is\_Exception (xa h)  $\wedge r = Inr (ex (xa h)) \longrightarrow P h r$ )"
proof (rule ccpo.admissibleI)
  fix A :: "('a  $\Rightarrow$  ('b, 'c) result) set"
  assume ch: "Complete_Partial_Order.chain (fun_ord result_ord) A"
  and IH: " $\forall xa \in A. \forall h r. is\_Normal (xa h) \wedge r = Inl (normal (xa h)) \vee is\_Exception (xa h) \wedge r = Inr$ 
  (ex (xa h))  $\longrightarrow P h r$ "
  from ch have ch': " $\bigwedge x. Complete\_Partial\_Order.chain result\_ord \{y. \exists f \in A. y = f x\}$ " by (rule
  chain_fun)
  show " $\forall h r. is\_Normal (fun_lub result_lub A h) \wedge r = Inl (normal (fun_lub result_lub A h)) \vee$ 
  is\_Exception (fun_lub result_lub A h)  $\wedge r = Inr (ex (fun_lub result_lub A h)) \longrightarrow P h r$ "
  proof (intro allI impI)
    fix h r assume "is\_Normal (fun_lub result_lub A h)  $\wedge r = Inl (normal (fun_lub result_lub A h)) \vee$ 
  is\_Exception (fun_lub result_lub A h)  $\wedge r = Inr (ex (fun_lub result_lub A h))$ "
    then show "P h r"
    proof
      assume "is\_Normal (fun_lub result_lub A h)  $\wedge r = Inl (normal (fun_lub result_lub A h))$ "
      with flat_lub_in_chain[OF ch'] this[unfolded fun_lub_def]
      show "P h r" using IH
      by (smt (verit) Collect_cong mem_Collect_eq result.case_eq_if result.disc_eq_case(3))
    next
      assume "is\_Exception (fun_lub result_lub A h)  $\wedge r = Inr (ex (fun_lub result_lub A h))$ "
      with flat_lub_in_chain[OF ch'] this[unfolded fun_lub_def]
      show "P h r" using IH
      by (smt (verit) Collect_cong mem_Collect_eq result.case_eq_if result.disc_eq_case(3))
    qed
  qed
qed
```

qed

```

lemma admissible_sm:
  "sm.admissible ( $\lambda f. \forall x h r. \text{effect } (f x) h r \longrightarrow P x h r$ )"
proof (rule admissible_fun[OF sm_interpretation])
  fix x
  show "ccpo.admissible sm_lub sm_ord ( $\lambda a. \forall h r. \text{effect } a h r \longrightarrow P x h r$ )"
    unfolding sm_ord_def sm_lub_def
  proof (intro admissible_image partial_function_lift flat_interpretation)
    show "ccpo.admissible (fun_lub result_lub) (fun_ord result_ord) (( $\lambda a. \forall h r. \text{effect } a h r \longrightarrow P x h r$ )
  r) o create)"
    unfolding comp_def effect_def execute_create
    by (rule sm_step_admissible)
  qed (auto simp add: execute_simps)
qed

```

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  $\Rightarrow$  'c" and
    U :: "'c  $\Rightarrow$  'b  $\Rightarrow$  ('a, 'e, 's) state_monad" and
    C :: "('b  $\Rightarrow$  ('a, 'e, 's) state_monad)  $\Rightarrow$  'c" and
    P :: "'b  $\Rightarrow$  's  $\Rightarrow$  'a  $\times$  's + 'e  $\times$  's  $\Rightarrow$  bool"
  assumes mono: " $\bigwedge x. \text{monotone } (\text{fun\_ord } sm\_ord) sm\_ord (\lambda f. U (F (C f)) x)$ "
  assumes eq: " $f \equiv C (\text{ccpo.fixp } (\text{fun\_lub } sm\_lub) (\text{fun\_ord } sm\_ord) (\lambda f. U (F (C f))))$ "
  assumes inverse2: " $\bigwedge f. U (C f) = f$ "
  assumes step: " $\bigwedge f x h r. (\bigwedge x h r. \text{effect } (U f x) h r \Longrightarrow P x h r) \Longrightarrow \text{effect } (U (F f) x) h r \Longrightarrow P x h r$ "
  assumes defined: "effect (U f x) h r"
  shows "P x h r"
  using step defined sm.fixp_induct_uc[of U F C, OF mono eq inverse2 admissible_sm, of P]
  unfolding effect_def execute_create by blast

```

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

```

declaration <
  Partial_Function.init "sm"
  term <sm.fixp_fun>
  term <sm.mono_body>
  @{thm sm.fixp_rule_uc}
  @{thm sm.fixp_induct_uc}
  (SOME @{thm fixp_induct_sm})
>

```

2.14.9 Monotonicity Results

```

abbreviation "mono_sm  $\equiv$  monotone (fun_ord sm_ord) sm_ord"

```

```

lemma execute_bind_case:
  "execute (f  $\ggg$  g) h = (case (execute f h) of
    Normal (x, h')  $\Rightarrow$  execute (g x) h' | Exception e  $\Rightarrow$  Exception e | NT  $\Rightarrow$  NT)"
  by (simp add: bind_def execute_simps)

```

```

lemma bind_mono [partial_function_mono,mono]:
  assumes mf: "mono_sm B" and mg: " $\bigwedge y. \text{mono\_sm } (\lambda f. C y f)$ "
  shows "mono_sm ( $\lambda f. B f \ggg (\lambda y. C y f)$ )"
proof (rule monotoneI)
  fix f g :: "'a  $\Rightarrow$  ('b, 'c, 'd) state_monad" assume fg: "sm.le_fun f g"
  from mf
  have 1: "sm_ord (B f) (B g)" by (rule monotoneD) (rule fg)
  from mg
  have 2: " $\bigwedge y'. sm\_ord (C y' f) (C y' g)$ " by (rule monotoneD) (rule fg)

  have "sm_ord (B f  $\ggg$  ( $\lambda y. C y f$ )) (B g  $\ggg$  ( $\lambda y. C y f$ ))" (is "sm_ord ?L ?R")
  proof (rule sm_ordI)

```

```

fix h
from 1 show "execute ?L h = NT  $\vee$  execute ?L h = execute ?R h"
  by (rule sm_ordE[where h = h]) (auto simp: execute_bind_case)
qed
also
have "sm_ord (B g  $\ggg$  ( $\lambda y'. C y' f$ )) (B g  $\ggg$  ( $\lambda y'. C y' g$ ))" (is "sm_ord ?L ?R")
proof (rule sm_ordI)
  fix h
  show "execute ?L h = NT  $\vee$  execute ?L h = execute ?R h"
  proof (cases "execute (B g) h")
    case (n a s)
    then have "execute ?L h = execute (C a f) s" "execute ?R h = execute (C a g) s"
      by (auto simp: execute_bind_case)
    with 2[of a] show ?thesis by (auto elim: sm_ordE)
  next
  case (e e)
  then show ?thesis by (simp add: execute_bind_case)
  next
  case t
  then have "execute ?L h = NT" by (auto simp: execute_bind_case)
  thus ?thesis ..
  qed
  qed
  finally (sm.leq_trans)
  show "sm_ord (B f  $\ggg$  ( $\lambda y. C y f$ )) (B g  $\ggg$  ( $\lambda y'. C y' g$ ))" .
qed

lemma throw_monad_mono[mono]: "mono_sm ( $\lambda_. \text{throw } e$ )"
  by (simp add: monotoneI sm_ordI)

lemma return_monad_mono[mono]: "mono_sm ( $\lambda_. \text{return } x$ )"
  by (simp add: monotoneI sm_ordI)

lemma option_monad_mono[mono]: "mono_sm ( $\lambda_. \text{option } E x$ )"
  by (simp add: monotoneI sm_ordI)

definition exc:: "('a, 'b, 'c) state_monad  $\Rightarrow$  ('a, 'b, 'c) state_monad"
  where "exc m  $\equiv$  create ( $\lambda s. \text{case execute } m s \text{ of Normal } (v, s') \Rightarrow \text{Normal } (v, s')$ 
    | Exception (e, s')  $\Rightarrow$  Exception (e, s)
    | NT  $\Rightarrow$  NT)"

lemma exc_mono[mono]:
  fixes m:: "('b  $\Rightarrow$  ('c, 'e, 'f) state_monad)  $\Rightarrow$  ('x, 'y, 'z) state_monad"
  assumes mf: "mono_sm ( $\lambda \text{call}. (m \text{ call})$ )"
  shows "mono_sm ( $\lambda \text{call}. (\text{exc } (m \text{ call}))$ )"

proof (rule monotoneI)
  fix f g :: "'b  $\Rightarrow$  ('c, 'e, 'f) state_monad"
  assume fg: "sm.le_fun f g"
  then have 1: "sm_ord (m f) (m g)" using mf by (auto dest: monotoneD)
  show "sm_ord (exc (m f)) (exc (m g))"
  proof (rule sm_ordI)
    fix h
    show "execute (exc (m f)) h = NT  $\vee$  execute (exc (m f)) h = execute (exc (m g)) h"
    proof (rule sm_ordE[OF 1, of h])
      assume "execute (m f) h = NT"
      then show "execute (exc (m f)) h = NT  $\vee$  execute (exc (m f)) h = execute (exc (m g)) h" unfolding
        exc_def by (simp add: execute_simps)
    next
      assume "execute (m f) h = execute (m g) h"
      then show "execute (exc (m f)) h = NT  $\vee$  execute (exc (m f)) h = execute (exc (m g)) h" unfolding
        exc_def by (simp add: execute_simps)
    qed
  qed
  qed

```

qed

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"
  unfolding case_memory_def using assms by (simp split: option.split mdata.split)
```

```
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'')"
  using assms
  by (cases "to_nat i", auto simp add:case_memory_def split:option.split_asm mdata.split_asm)
```

```

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"
  using assms
  by (cases "to_nat i",
      auto simp add:case_memory_def nth_safe_def split: mdata.split_asm if_split_asm)

lemma marray_lookup_prefix:
  assumes "marray_lookup m xs l = Some x"
    and "prefix m m'"
  shows "marray_lookup m' xs l = Some x"
  using assms
proof (induction xs arbitrary: l)
  case Nil
  then show ?case by simp
next
  case (Cons i xs')
  then show ?case
  proof (cases xs')
    case Nil
    then show ?thesis using Cons
      apply (auto simp add:case_memory_def)
      apply (case_tac "m $ l", auto)
      apply (case_tac "a", auto)
      using nth_safe_prefix by fastforce
  next
    case c: (Cons i' "is")
    then obtain 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 x"
    using marray_lookup_obtain_multi Cons(2) by blast
    moreover from * have "marray_lookup m' (i'#is) l' = Some x" using Cons(1,3) c by simp
    ultimately show ?thesis using Cons(3) c
      apply (auto simp add:case_memory_def)
      using nth_safe_prefix by fastforce
  qed
qed

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"
  using assms
proof (induction xs arbitrary: y l rule: list_nonempty_induct)
  case (single x)
  then show ?case
  apply (cases ys, auto simp add:case_memory_def)
  apply (metis surj_pair)
  apply (cases "m$l", auto)
  apply (case_tac aa, auto)
  by (cases "to_nat x", auto)
next
  case (cons x xs)
  then obtain x' xs'
  where *: "marray_lookup m (x # x' # (xs' @ ys)) l = Some y"
    and **: "xs = x' # xs'"
  by (metis append_Cons neq_Nil_conv)

```

```

then obtain ns i'' l'
  where 1: "m $ l = Some (mdata.Array ns)"
        and 2: "to_nat x = Some i''"
        and 3: "ns $ i'' = Some l'"
        and 4: "marray_lookup m (x' # xs' @ ys) l' = Some y"
  using marray_lookup_obtain_multi[OF *] by blast
moreover from 1 2 3 4 obtain y
  where "marray_lookup m xs l' = Some y" using cons(2) **
  by fastforce
ultimately show ?case using **
  apply (auto simp add:case_memory_def)
  by (metis surj_pair)
qed

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))"
  using assms
proof (induction xs arbitrary: l)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  then obtain y ys' where *: "ys = y#ys'" by (meson list.exhaust)
  show ?case
  proof (cases xs)
    case Nil
    then show ?thesis using * by (auto simp add:case_memory_def nth_safe_def split: mdata.split)
  next
    case 1: Cons
    show ?thesis
    proof (cases "m $ l")
      case (Some md)
      then show ?thesis
      proof (cases md)
        case (Array ls)
        then have
          **: "marray_lookup m ((x # xs) @ ys) l = to_nat x ≧≧ ($) ls ≧≧ marray_lookup m (xs @ ys)"
            using 1 Some by (auto simp add:case_memory_def)
          show ?thesis
          proof (cases "to_nat x")
            case 2: (Some i)
            then show ?thesis
            proof (cases "ls $ i")
              case None
              then show ?thesis
              using Cons(1,3) 1 2 by (auto simp add:case_memory_def nth_safe_def split: mdata.split)
            next
              case 3: (Some l')
              then have
                "marray_lookup m (xs @ ys) l'
                = marray_lookup m xs l' ≧≧ (λa. case a of (l', ls, i) ⇒ ls $ i ≧≧ marray_lookup m ys)"
                  using Cons(1) * 1 by auto
              then show ?thesis using Some Array 1 2 3 by (auto simp add:case_memory_def)
            qed
          qed (auto simp add: 1 case_memory_def split:option.split mdata.split)
        qed (auto simp add: 1 case_memory_def)
      qed (auto simp add: 1 case_memory_def)
    qed
  qed

```

qed

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"
  using assms by simp
```

```
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"
  using assms
  by (cases "to_nat i", auto simp add:case_memory_def split:option.split_asm mdata.split_asm)
```

```
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'"
  using assms apply (auto)
  apply (cases "marray_lookup m (x # xs) l") by (auto)
```

```
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'"
  using assms that
  apply (cases "is", auto simp add:case_memory_def)
  apply (cases "m$l", auto)
  apply (case_tac a, auto)
  apply (cases "to_nat i", auto)
  apply (cases "m$l", auto)
  apply (case_tac aa, auto)
  apply (cases "to_nat i", auto)
  by (case_tac "x2$aa", auto)
```

```
lemma mlookup_start_some:
  assumes "mlookup m (iv#is) l = Some l'"
  shows "∃x. m$l = Some x"
  using assms
  proof (cases rule:mlookup_obtain_nempty2)
  case (1 ls i' l'')
  then show ?thesis by blast
  qed
```

```
lemma mlookup_same:
  assumes "xs ≠ []"
    and "m $ l1 = m $ l2"
  shows "mlookup m xs l1 = mlookup m xs l2"
  using assms
  proof (induction xs arbitrary: l1 l2 rule: list_nonempty_induct)
  case (single x)
```

```

then show ?case
  apply (cases "m$l2", auto simp add: case_memory_def)
  by (case_tac a, auto)
next
case (cons x xs)
then obtain x' xs' where xs_def: "xs=x'#xs'"
  by (meson list.exhaust)
then show ?case
proof (cases "m $ l1")
  case None
  then have "m $ l2 = None" using cons(3) by simp
  then show ?thesis using None xs_def
    by (simp add: case_memory_def)
next
  case (Some a)
  then have "m $ l2 = Some a" using cons(3) by simp
  then show ?thesis using Some xs_def
    by (simp add: case_memory_def)
qed
qed

lemma mlookup_prefix_mlookup:
  assumes "mlookup m xs0 l = Some x0"
    and "prefix m m'"
  shows "mlookup m' xs0 l = Some x0"
  using assms
proof (cases xs0)
  case Nil
  then show ?thesis using assms by auto
next
  case (Cons x xs)
  then obtain a xs' i i'
  where *: "marray_lookup m (x#xs) l = Some (a, xs', i)"
    and "xs' $ i = Some i'"
    and "x0 = i'"
  using mlookup_obtain_nempty1 assms(1) by metis
  moreover have "marray_lookup m' (x#xs) l = Some (a, xs', i)"
    using marray_lookup_prefix[OF * assms(2)] by blast
  ultimately show ?thesis using assms Cons by auto
qed

lemma mlookup_append:
  "mlookup m (xs @ ys) l = mlookup m xs l  $\gg$  mlookup m ys"
proof (cases xs)
  case 1: Nil
  then show ?thesis
  proof (cases ys)
    case Nil
    then show ?thesis using 1 by (auto simp add: nth_safe_def)
  next
    case 2: (Cons _ ys')
    then show ?thesis using 1 by (cases ys', auto simp add: nth_safe_def case_memory_def)
  qed
next
case (Cons x xs')
then show ?thesis
proof (cases ys)
  case Nil
  then show ?thesis
  proof (cases xs')
    case 1: Nil
    then show ?thesis
    by (cases "to_nat x",
        auto simp add: Cons Nil case_memory_def nth_safe_def split: mdata.split)

```

```

next
  case 1: (Cons x' xs')
  then show ?thesis
  proof (cases "to_nat x")
    case None
    then show ?thesis
      using Nil 1 Cons by (auto simp add: case_memory_def nth_safe_def split: mdata.split)
  next
  case (Some a)
  then show ?thesis using Nil 1 Cons
    apply (auto simp add: case_memory_def nth_safe_def
      split: option.split mdata.split if_split_asm)
    apply (case_tac "marray_lookup m (x' # xs') (x2 ! a)")
    by (auto simp add: case_memory_def nth_safe_def
      split: option.split mdata.split if_split_asm)
  qed
qed
next
  case 1: (Cons _ ys')
  then have
    lhs: "mlookup m (xs @ ys) l
      = marray_lookup m xs l
         $\gg=$  ( $\lambda(l', ls, i). ls \$ i \gg=$  marray_lookup m ys)
         $\gg=$  ( $\lambda(uu, xs', i). xs' \$ i$ )"
    using marray_lookup_append[of xs ys m l] Cons by simp
  show ?thesis
  proof (cases "marray_lookup m xs l")
    case None
    then show ?thesis using Cons 1 lhs by simp
  next
  case (Some a)
  then show ?thesis
  proof (cases a)
    case (fields l' ls i)
    then show ?thesis
    proof (cases "ls $ i")
      case None
      then show ?thesis using lhs Cons Some fields by auto
    next
    case 2: (Some a)
    then show ?thesis
    proof (cases "m $ a")
      case _: None
      then have "marray_lookup m ys a = None"
        using 1 by (cases ys', auto simp add: case_memory_def)
      then show ?thesis using lhs Cons Some fields 1 2 by auto
    next
    case _: (Some a)
    then show ?thesis using lhs Cons Some fields 1 2 by auto
  qed
  qed
  qed
  qed
  qed
qed

```

3.4 Memory Update (Memory)

```

fun mupdate :: "'v::vtype list  $\Rightarrow$  location  $\times$  'v mdata  $\times$  'v memory  $\Rightarrow$  'v memory option" where
  "mupdate xs (l, v, m) = mlookup m xs l  $\gg=$  ( $\lambda l. list\_update\_safe m l v$ )"

```

```

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]"
using assms by (cases "mlookup m xs l", auto simp add: list_update_safe_def split:if_split_asm)

```

```

lemma mvalue_update_length:
  assumes "mupdate is (ml, v, m) = Some m'"
  shows "length m' = length m"
  by (metis assms length_list_update mvalue_update_obtain)

```

3.5 Memory Locations (Memory)

```

fun locations :: "'v mdata list  $\Rightarrow$  'v::vtype list  $\Rightarrow$  nat  $\Rightarrow$  nat fset option" where
  "locations m [] l = Some ({})"
| "locations m (i#is) l =
  case_memory m l
  (K None)
  ( $\lambda$ xs. to_nat i  $\gg$  ($) xs  $\gg$  ( $\lambda$ l'. locations m is l'  $\gg$  ( $\lambda$ ls. 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')"
  using assms
  apply (cases "to_nat i", auto simp add: case_memory_def split:option.split_asm mdata.split_asm)
  by (case_tac "locations m is (x2a ! a)", auto simp add: nth_safe_def split: if_split_asm)

```

```

lemma locations_l_in_L:
  assumes "locations m (i#is') l = Some L"
  shows "l  $\in$  L"
proof-
  from assms obtain 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'" using locations_obtain by blast
  then show ?thesis
  by (auto simp add: locations_obtain case_memory_def split:option.split_asm mdata.split_asm)
qed

```

```

lemma locations_same:
  assumes "locations m xs0 l = Some L"
  and " $\forall l \in L. m' $ l = m $ l"$ 
  shows "locations m' xs0 l = Some L"
  using assms
proof (induction xs0 arbitrary: l L)
  case Nil
  then show ?case
  by auto
next
  case (Cons i "is")
  then obtain 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')"
  using locations_obtain by blast
  moreover from ** have " $\forall l \in L'. m' $ l = m $ l"$ 

```

3 Stores and State

```

    by (simp add: Cons.prem(2))
  then have "locations m' is l' = Some L'" using Cons(1)[OF *] by blast
  ultimately show ?case using Cons(3)
    by (auto simp add: case_memory_def)
qed

```

```

lemma locations_append_subset:
  assumes "locations m (xs @ xs') l = Some L"
  obtains L'
  where "locations m xs l = Some L'"
    and "L' | $\subseteq$ | L"
  using assms
proof (induction xs arbitrary: L l)
  case Nil
  then show ?case using locations_l_in_L by auto
next
  case (Cons i "is")
  then obtain xs i' l' ls'
    where "m$l = Some (mdata.Array xs)"
      and "to_nat i = Some i'"
      and "xs $ i' = Some l'"
      and "locations m (is @ xs') l' = Some ls'"
      and "L = (finsert l ls')"
    using locations_obtain[of m i "is @xs'" l L] by (metis append_Cons)
  then show ?case using Cons(1)[of l' ls'] Cons(2) Cons(3)
    by (cases "locations m is l'") (auto simp add: case_memory_def)
qed

```

```

lemma locations_prefix_locations:
  assumes "locations m xs0 l = Some L"
    and "prefix m m'"
  shows "locations m' xs0 l = Some L"
  using assms
proof (induction xs0 arbitrary: l L)
  case Nil
  then show ?case
    by auto
next
  case (Cons i "is")
  then obtain 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')"
    using locations_obtain by blast
  moreover have "locations m' is l' = Some L'" using Cons(1)[OF * Cons(3)] .
  ultimately show ?case using assms(2)
    apply (auto simp add: case_memory_def)
    using nth_safe_prefix by fastforce
qed

```

```

lemma locations_subs_loc:
  assumes "locations m xs0 l = Some L"
  shows "fset L  $\subseteq$  loc m"
  using assms
proof (induction xs0 arbitrary: l L)
  case Nil
  then show ?case
    by auto
next
  case (Cons i "is")
  then obtain 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 = (finset l L')"
  using locations_obtain by blast
moreover have "fset L'  $\subseteq$  loc m" using Cons(1)[OF **] .
moreover from * have "l  $\in$  loc m" unfolding nth_safe_def apply (simp split:if_split_asm)
  by (simp add: loc_def)
ultimately show ?case by blast
qed

```

3.6 Locations and Array Lookup (Memory)

```

lemma locations_marray_lookup_same:
  assumes "locations m1 is l = Some L"
    and " $\bigwedge l. l \in L \implies m1 \$ l = m2 \$ l$ "
  shows "marray_lookup m1 is l = marray_lookup m2 is l"
  using assms
proof (induction "is" arbitrary: l L)
  case Nil
  then show ?case by simp
next
  case (Cons i is')
  from Cons(3) Cons.prem(1) have 0: "m1$l = m2$l" using locations_l_in_L by blast
  from Cons(2) obtain xs i' l' ls'
    where 1: "m1$l = Some (mdata.Array xs)"
      and 2: "to_nat i = Some i'"
      and 3: "xs $ i' = Some l'"
      and 4: "locations m1 is' l' = Some ls'"
      and 5: "L = (finset l ls')"
  using locations_obtain by blast
  from Cons(1)[OF 4] have 6: "marray_lookup m1 is' l' = marray_lookup m2 is' l'"
  by (simp add: Cons.prem(2) 5)
  show ?case
proof (cases is')
  case Nil
  then show ?thesis using 0 by (simp add:case_memory_def)
next
  case (Cons a list)
  then show ?thesis using 0 1 2 3 6 by (cases "m2$l", auto simp add:case_memory_def)
qed
qed

```

```

lemma marray_lookup_in_locations:
  assumes "marray_lookup m is l = Some (l'', xs, i)"
    and "locations m is l = Some L"
  shows "l''  $\in$  L"
  using assms
proof (induction "is" arbitrary:l L)
  case Nil
  then show ?case by simp
next
  case (Cons a xs')
  then show ?case
proof (cases xs')
  case Nil
  then show ?thesis using Cons
    apply (auto simp add:case_memory_def split:option.split_asm mdata.split_asm)
    apply (cases "vtype_class.to_nat a", auto)
    by (cases "xs$i", auto)
next
  case c: (Cons x xs'')
  then obtain l' xs''' x0' where b1: "marray_lookup m xs' l' = Some (l'', xs, i)"

```

```

and b2: "m$l = Some (mdata.Array xs'''"
and b3: "to_nat a = Some x0'"
and b4: "xs''' $ x0' = Some l'"
using Cons(2) c
apply (auto simp add:case_memory_def split:option.split_asm mdata.split_asm)
apply (cases "vtype_class.to_nat a", auto)
apply (case_tac "x2a $ aa")
by auto

moreover from Cons(3) obtain L' where b2: "locations m xs' l' = Some L'" and "L' |⊆| L"
  apply (auto simp add:case_memory_def split:option.split_asm mdata.split_asm)
  apply (cases "vtype_class.to_nat a",simp)
  apply (case_tac "x2a $ aa", simp)
  apply (case_tac " locations m xs' aaa")
  using b2 b3 b4 by auto
ultimately have "l'' |∈| L'" using Cons(1) Cons by blast
then show ?thesis using <L' |⊆| L> by blast
qed
qed

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 -
  from assms(2) have "∧l. l |∈| L ⇒ m $ l = m[l':=v'] $ l"
  by (metis length_list_update nth_list_update_neq nth_safe_def)
  then show ?thesis using locations_marray_lookup_same[OF assms(1)] by metis
qed

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"
  using assms
proof (induction xs arbitrary:l)
  case Nil
  then show ?case by simp
next
  case (Cons i' "is")
  show ?case
  proof (cases "is")
  case Nil
  then show ?thesis using Cons(2,3) apply (auto simp add:case_memory_def)
  apply (case_tac "m$l",auto)
  apply (case_tac a,auto)
  by (case_tac "to_nat i'",auto)
  next
  case c: (Cons i'' is')
  then obtain 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 (l0, xs', i)"
  using marray_lookup_obtain_multi[of m i' i'' is' l] Cons(2) Cons(3) by blast
  moreover from * obtain L where "locations m is l' = Some L" using Cons c by auto
  ultimately show ?thesis using Cons by (auto simp add:case_memory_def)
  qed
qed
qed

lemma marray_lookup_in_locations2:
  assumes "xs ≠ []"
  and "ys ≠ []"
  and "marray_lookup m xs l = Some (l0, xs0, i0)"

```

```

    and "xs0 $ i0 = Some l1"
    and "locations m (xs@ys) l = Some L"
  shows "l1 |∈| L"
using assms
proof (induction "xs" arbitrary: m l L rule: list_nonempty_induct)
case (single i)
then obtain xs i''
  where "m $ l = Some (mdata.Array xs)"
  and "to_nat i = Some i''"
  and "(l0, xs0, i0) = (l, xs, i'')"
  using marray_lookup_obtain_single by blast
then show ?case using single
  apply (case_tac "locations m ys l1", auto simp add:case_memory_def)
  by (metis list.exhaust_sel locations_l_in_L)
next
case (cons i "is")
then obtain i' is' where is_def: "is = i' # is'"
  by (meson list.exhaust)
with cons(4) have *: "marray_lookup m (i # i' # is') l = Some (l0, xs0, i0)" by simp
obtain xs i'' l'
where "m $ l = Some (mdata.Array xs)"
  and "to_nat i = Some i''"
  and "xs $ i'' = Some l'"
  and l4: "marray_lookup m is l' = Some (l0, xs0, i0)"
  using marray_lookup_obtain_multi[OF *] is_def by auto
moreover from cons(6) have *: "locations m (i # (is @ ys)) l = Some L" by simp
ultimately obtain L' where "locations m (is @ ys) l' = Some L'"
  and "L = (finsert l L')"
  using locations_obtain[OF *] by force
with cons(2)[OF cons(3) l4 cons(5)] show ?case by simp
qed

```

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'"
using assms
proof (cases xs)
case Nil
then have "l = l'"
  using mlookup_obtain_empty assms(1) by blast
then have "mlookup (m[l'':=v]) [] l = Some l'"
  by (cases "m[l'':=v]$l", auto simp add: nth_safe_def split:if_split_asm)
then show ?thesis using Nil(1) by simp
next
case (Cons x xs'')
then obtain a xs' i i'
  where "marray_lookup m (x # xs'') l = Some (a, xs', i)"
  and "xs' $ i = Some i'"
  and "l' = i'"
  using mlookup_obtain_empty1[of m x xs'' l "l'"] assms(1) by metis
moreover have "marray_lookup m xs l = marray_lookup (m[l'':=v]) xs l"
  using marray_lookup_update_same[OF assms(2) assms(3)] by simp
ultimately show ?thesis using Cons(1) assms
  by (cases "marray_lookup (m[i' := v]) (x # xs'') l")
  (auto simp add: nth_safe_def split:if_split_asm)
qed

```

```

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

```

3 Stores and State

```

using assms
proof (cases xs0)
  case Nil
  then show ?thesis by simp
next
  case (Cons x xs)

  then obtain a xs' i i'
  where "marray_lookup m (x#xs) l = Some (a, xs', i)"
    and "xs' $ i = Some i'"
    and "l' = i'" using mlookup_obtain_nempty1 assms(1)
    by metis

  then obtain L where "locations m (x#xs) l = Some L" using marray_lookup_locations_some by blast
  then show ?thesis using Cons by blast
qed

```

```

lemma mlookup_update_same_nempty:
  assumes "mlookup m (x#xs) l1 = Some l1'"
    and "locations m (x#xs) l1 = Some L"
    and " $\neg (l2 \in L)$ "
  shows "mlookup (m[l2:=v']) (x#xs) l1 = mlookup m (x#xs) l1"
  using mlookup_update_val[OF assms(1,2)]
proof -
  from assms(1) obtain a xs' i i'
  where "marray_lookup m (x # xs) l1 = Some (a, xs', i)"
    and "xs' $ i = Some i'"
    and "l1' = i'"
  using mlookup_obtain_nempty1[of m x xs l1 "l1'"] assms(1) by metis
  moreover have "marray_lookup m (x#xs) l1 = marray_lookup (m[l2:=v']) (x#xs) l1"
  using marray_lookup_update_same[OF assms(2) assms(3)] by simp
  ultimately show ?thesis using Cons(1)
  by (auto simp add: nth_safe_def split: if_split_asm)
qed

```

```

lemma mlookup_in_locations:
  assumes "ys  $\neq$  []"
    and "mlookup m xs l = Some l'"
    and "locations m (xs@ys) l = Some L"
  shows "l'  $\in$  L"
  using assms
proof (cases xs)
  case Nil
  then show ?thesis using assms
  by (metis list.exhaust locations_l_in_L mlookup_obtain_empty self_append_conv2)
next
  case (Cons a list)
  then show ?thesis
  by (metis assms(1,2,3) list.distinct(1) marray_lookup_in_locations2 mlookup_obtain_nempty1)
qed

```

```

lemma mlookup_access_same:
  assumes "locations m1 xs l = Some L"
    and "mlookup m1 xs l = Some l'"
    and " $\bigwedge l. l \in L \implies m1 \$ l = m2 \$ l$ "
    and "m1 $ l' = m2 $ l'"
  shows "mlookup m2 xs l  $\gg$  ($) m2 = m1 $ l'"
proof (cases xs)
  case Nil
  then show ?thesis using assms by simp
next
  case (Cons x' xs')
  then have "mlookup m2 xs l = Some l'"
  using locations_marray_lookup_same[OF assms(1,3)] assms(2) by simp

```

```

then show ?thesis using assms(4) by auto
qed

```

```

lemma mlookup_same_locations:
  assumes "mlookup m1 xs l = Some l'"
    and "locations m1 xs l = Some L"
    and " $\forall l \in L. m1 \$ l = m2 \$ l$ "
  shows "mlookup m2 xs l = Some l'"
  using assms
proof (cases xs)
  case Nil
  then show ?thesis
    using assms(1) by auto
next
  case (Cons x xs')
  then obtain a xs'' i i'
  where "marray_lookup m1 (x#xs') l = Some (a, xs'', i)"
    and "xs'' $ i = Some i'"
    and "l' = i'"
  using mlookup_obtain_empty1 assms(1) by blast
  then have "marray_lookup m1 xs l = marray_lookup m2 xs l"
    using locations_marray_lookup_same[OF assms(2), of m2] assms(3) by blast
  then show ?thesis using assms(1) local.Cons by fastforce
qed

```

```

lemma mlookup_append_same:
  assumes "ys  $\neq$  []"
    and "mlookup m xs1 l1 = Some l1'"
    and "m $ l1' = Some l1''"
    and "mlookup m xs2 l2  $\gg=$  ($) m = Some l1''"
  shows "mlookup m (xs1 @ ys) l1 = mlookup m (xs2 @ ys) l2" (is "?LHS=?RHS")
proof -
  from assms(4) obtain l2' where ls'_def: "mlookup m xs2 l2 = Some l2'" and l1''_def: "m $ l2' = Some l1''"
  by (meson bind_eq_Some_conv)
  have "?LHS = mlookup m xs1 l1  $\gg=$  mlookup m ys" using mlookup_append by blast
  also have "... = mlookup m ys l1'" using assms by simp
  also have "... = mlookup m ys l2'" using mlookup_same[OF assms(1)] assms(3) l1''_def by auto
  also have "... = mlookup m xs2 l2  $\gg=$  mlookup m ys" using ls'_def by simp
  also have "... = ?RHS" using mlookup_append by metis
  finally show ?thesis .
qed

```

```

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 " $\forall l \in L. m0 \$ l = m1 \$ l$ "
  shows "locations m1 xs l1 = Some (finsert l1 L)"
proof -
  from assms(1) have "xs  $\neq$  []" by simp
  then show ?thesis using assms
proof (induction xs arbitrary: l0 l1 L l x xs' rule: list_nonempty_induct)
  case (single x)
  then show ?case apply (simp add:case_memory_def split:option.split_asm mdata.split_asm)
    by (case_tac "vtype_class.to_nat x", auto)
next
  case (cons i "is")
  moreover obtain as i'
  where ls_def: "m0 $ l0 = Some (Array as)"
    and i'_def: "vtype_class.to_nat x = Some i'" and l_def: "as $ i' = Some l"
  using mlookup_obtain_single[OF cons(5)] by blast
  moreover from ls_def have as_def: "m1 $ l1 = Some (Array as)" using cons by argo

```

```

moreover obtain LL where LL_def: "locations m1 is l = Some LL"
  by (metis cons.prem1(1,4,5) list.inject locations_same)
ultimately have "locations m1 (i # is) l1 = Some (finsert l1 LL)" by (auto simp add:
case_memory_def)
show ?case
proof (cases xs')
  case Nil
  then show ?thesis using cons by simp
next
  case (Cons x' xs'')
  moreover have "m0 $ l = m1 $ l"
    using calculation cons.prem1(4,5) locations_l_in_L by blast
  moreover obtain l2 where "mlookup m0 [x'] l = Some l2" using cons(6) Cons apply (auto simp
add:case_memory_def split:option.split_asm mdata.split_asm)
  apply (case_tac "vtype_class.to_nat x'",auto)
  by (case_tac " x2a $ a",auto)
  moreover obtain L2 where "locations m0 xs'' l2 = Some L2" and L2_def: "L = (finsert l L2)"
using cons(6) Cons apply (auto simp add:case_memory_def split:option.split_asm mdata.split_asm)
  apply (case_tac "vtype_class.to_nat x'",auto)
  apply (case_tac " x2a $ a",auto)
  apply (case_tac "locations m0 xs'' aa",auto)
  by (metis calculation(3) mdata.inject(2) mlookup_obtain_single option.sel)
  moreover have " $\forall l \in L2. m0 \$ l = m1 \$ l$ "
    by (simp add: L2_def cons.prem1(5))
  ultimately have "locations m1 is l = Some (finsert l L2)" using cons(2)[of x' xs'' l l L2]
    using cons.prem1(1) by blast
  then show ?thesis using as_def LL_def i'_def cons(3) l_def L2_def by (simp add:case_memory_def)
qed
qed
qed

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 ll"
    and "locations m0 xs l = Some L0"
    and " $\forall l \in L0. m0 \$ l = m1 \$ l$ "
    and "locations m0 ys' ll = Some L1"
    and " $\forall l \in L1. m0 \$ l = m1 \$ l$ "
  shows "locations m1 (xs @ ys) l = Some (finsert l' L0 | $\cup$ | L1)"
  using assms
proof (induction xs arbitrary: l l' L0)
  case Nil
  then have "locations m1 ys l = Some (finsert l L1)"
    using locations_union_nth[of ys y ys' m0 l'' m1 l ll L1] by auto
  moreover from Nil have "L0 = {}" by simp
  ultimately show ?case using Nil.prem1(2) by force
next
  case (Cons i "is")
  obtain ls i' l''
    where ls_def: "m0 $ l = Some (Array ls)"
      and i'_def: "vtype_class.to_nat i = Some i'"
      and l''_def: "ls $ i' = Some l''"
      and l'_def: "mlookup m0 is l'' = Some l'"
  using mlookup_obtain_empty2[OF Cons(3)] by blast
  moreover from ls_def i'_def l''_def obtain L'
  where L'_def: "locations m0 is l'' = Some L'"
    and L0_def: "L0 = (finsert l L')"
  using locations_obtain[OF Cons(6)] by force
  moreover have "locations m1 (is @ ys) l'' = Some (finsert l' L' | $\cup$ | L1)"
    using Cons(1)[OF Cons(2) l'_def Cons(4,5) L'_def _ Cons(8,9)] Cons(7) L0_def by auto
  ultimately show ?case using Cons(7) by (auto simp add:case_memory_def split:option.split
mdata.split)

```

qed

```

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 | $\cup$ | L1)"
  using assms
proof (induction xs arbitrary: l l' L0)
  case Nil
  then show ?case
    by simp
next
  case (Cons i "is")
  obtain ls i' l''
    where ls_def: "m $ l = Some (Array ls)"
      and i'_def: "vtype_class.to_nat i = Some i'"
      and l''_def: "ls $ i' = Some l''"
      and l'_def: "mlookup m is l'' = Some l'"
  using mlookup_obtain_nempty2[OF Cons(2)] by blast
  moreover from ls_def i'_def l''_def obtain L'
  where L'_def: "locations m is l'' = Some L'"
    and "L0 = (finsert l L')"
  using locations_obtain[OF Cons(3)] by force
  moreover have "locations m (is @ ys) l'' = Some (L' | $\cup$ | L1)"
    using Cons(1)[OF l'_def L'_def Cons(4)] by simp
  ultimately show ?case by (auto simp add:case_memory_def)
qed

```

```

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 | $\subseteq$ | L0"
proof -
  from assms(1) obtain L where "locations m xs l = Some L" using locations_append_subset[OF assms(2)]
  by metis
  then show ?thesis using locations_union_mlookup[OF assms(1) _ assms(3)] assms(2) by simp
qed

```

```

proposition is_none_mlookup_locations:
  assumes " $\neg$  Option.is_none (mlookup m xs l)"
  shows " $\neg$  Option.is_none (locations m xs l)"
  using assms
proof (induction xs arbitrary:l)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  then obtain aa where a_def: "mlookup m (x # xs) l = Some aa"
    by (metis Option.is_none_def option.exhaust)
  then obtain a xs' i i'
  where a1: "marray_lookup m (x#xs) l = Some (a, xs', i)"
    and a2: "xs' $ i = Some i'"
    and a4: "aa = i'"
  using mlookup_obtain_nempty1 by metis
  then show ?case
proof (cases xs)
  case Nil
  then show ?thesis using a1 a2 a4 Cons
    apply (auto simp add:case_memory_def split:option.split_asm mdata.split_asm)
    apply (cases "vtype_class.to_nat x") by auto
next
  case c: (Cons a' list)

```

```

then obtain l' xs'' x0' where b1: "marray_lookup m xs l' = Some (a, xs', i)"
  and "m$l = Some (mdata.Array xs'')"
  and "to_nat x = Some x0'"
  and "xs'' $ x0' = Some l'"
  using a1 a4 Cons(2)
  apply (auto simp add:case_memory_def split:option.split_asm mdata.split_asm)
  apply (cases "vtype_class.to_nat x") apply auto
  apply (case_tac "x2a $ ab") by auto
moreover have "¬ Option.is_none (locations m xs l')" using a1 c Cons.IH Cons.prems b1 by auto
moreover from a1 c obtain
  xs'' where "m$l = Some (mdata.Array xs'')" using marray_lookup_obtain_multi by blast
ultimately show ?thesis
  apply (simp add:case_memory_def)
  using Option.is_none_def by fastforce
qed
qed

```

```

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''"
  using assms
proof (induction xs arbitrary:l0 L)
  case Nil
  then show ?case by simp
next
  case (Cons x xs')
  then obtain ls i' l'
  where ls_def: "m$l0 = Some (mdata.Array ls)"
  and i'_def: "to_nat x = Some i'"
  and l'_def: "ls $ i' = Some l'"
  and l1_def: "mlookup m xs' l' = Some l1"
  using mlookup_obtain_nempty2[OF Cons(3)] by blast
  moreover from ls_def i'_def l'_def obtain L'
  where L'_def: "locations m (xs' @ ys) l' = Some L'"
  and "L = (finsert l0 L')"
  using locations_obtain[of m x "xs' @ ys" l0] using Cons(2) by force
  moreover from Cons(1)[OF L'_def l1_def]
  obtain L'' L'''
  where "locations m xs' l' = Some L''"
  and "locations m ys l1 = Some L'''"
  and "L = L'' |∪| L'''" by auto
  ultimately show ?case by (auto simp add: case_memory_def)
qed

```

```

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"
  using assms apply (auto simp add: case_memory_def split:option.split_asm mdata.split_asm)
  apply (cases "vtype_class.to_nat z", auto)
  by (cases "locations m0 zs l1", auto)

```

```

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"
  using assms
proof -
  from assms have "mlookup m1 ys l1 = Some l1'" using mlookup_same_locations[OF assms(1)]

```

```

  by (metis mlookup_locations_some option.sel)
moreover from assms have "mlookup m1 zs l0' = Some l2" using mlookup_same_locations[OF assms(6)]
  by (metis mlookup_locations_some option.sel)
moreover have "m1 $ l1' = m1 $ l0'"
  by (metis assms(2,3,4,6) locations.elims locations_l_in_L mlookup_locations_some option.sel)
ultimately show ?thesis
  by (metis assms(3) bind.bind_lunit mlookup_append mlookup_same)
qed

```

```

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|})))"
by pat_completeness auto
termination
  apply (relation "measure (λ(s,m,_). card ({0..length m} - fset s))")
  using card_less_card by simp_all

```

```

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"
  using assms
  by (cases "m$l", auto split:if_split_asm mdata.split_asm simp add: case_memory_def)

```

```

lemma range_safe_subs:
  assumes "range_safe s m l = Some X"
  shows "l |∈| X"
  using assms
proof (cases rule: range_safe_obtains)
  case (1 v)
  then show ?thesis by simp
next
  case (2 xs)
  then show ?thesis using fold_some by force
qed

```

```

lemma range_safe_subs2:
  assumes "range_safe s m l = Some X"
  shows "fset X ⊆ loc m"
  using assms
proof (induction l arbitrary: X rule:range_safe.induct[where ?a0.0=s])
  case (1 s m l)

```

```

from 1(2) show ?case
proof (cases rule:range_safe_obtains)
  case 2: (1 v)
  show ?thesis
  proof
    fix x assume "x |∈| X"
    then have "l = x" using 2 by simp
    moreover have "l ∈ loc m" using 2(2)
      unfolding loc_def nth_safe_def by (simp split:if_split_asm)
    ultimately show "x ∈ loc m" by simp
  qed
next
case (2 xs)
moreover have "l ∈ loc m"
  by (simp add: 2(2) loc_def nth_safe_length)
moreover have "∀x ∈ set xs. ∀L. range_safe (finsert l s) m x = Some L → fset L ⊆ loc m"
  using 1(1)[OF 2(1,2)] by simp
ultimately show ?thesis using fold_subs[of _ "range_safe (finsert l s) m"] by blast
qed
qed

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"
  using assms
proof -
  from assms(1) have "l |∉| s" by auto
  with assms(1,2) have "fold (λx y. y ≫= (λy'. range_safe (finsert l s) m x ≫= (λl. Some (l |∪| y')))) xs (Some {l|}) =
    Some L" by (simp add:case_memory_def)
  then show ?thesis by (metis assms(3) fold_some_subs that)
qed

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"
  using assms
proof (induction l arbitrary: L s' rule:range_safe.induct[where ?a0.0=s])
  case (1 s m l)
  from 1(2) show ?case
  proof (cases rule:range_safe_obtains)
    case _: (1 v)
    then show ?thesis using 1(3) by (auto simp add:case_memory_def)
  next
  case (2 xs)
  moreover from 2(1) 1(3) have "l |∉| s'" using "1.prem1" range_safe_subs by blast
  moreover have
    "fold (λx y. y ≫= (λy'. range_safe (finsert l s) m x ≫= (λl. Some (l |∪| y')))) xs (Some {l|})
    = fold (λx y. y ≫= (λy'. range_safe (finsert l s') m x ≫= (λl. Some (l |∪| y')))) xs (Some
    {l|})"
  proof (rule fold_same)
    show "∀x∈set xs. range_safe (finsert l s) m x = range_safe (finsert l s') m x"
  proof
    fix x assume "x ∈ set xs"
    moreover from 'x ∈ set xs' obtain y
      where "range_safe (finsert l s) m x = Some y"
        and "y |⊆| L"
      using fold_some_subs[OF 2(3)] by blast
    moreover from 1(3) have "∀l |∈| finsert l s' |-| finsert l s. l |∉| y"
      using 'y |⊆| L' by blast
    ultimately show "range_safe (finsert l s) m x = range_safe (finsert l s') m x"
  qed
  qed

```

```

    using 1(1)[OF 2(1,2), of x _ _ y "fininsert l s'"] by simp
  qed
  qed
  ultimately show ?thesis by (auto simp add:case_memory_def)
  qed
  qed

lemma range_safe_same:
  assumes "range_safe s m l = Some L"
    and " $\forall l' \in |L. m' \$ l' = m \$ l'$ "
  shows "range_safe s m' l = Some L"
  using assms
proof (induction l arbitrary: L rule:range_safe.induct)
  case (1 s m' l)
  from 1(2) show ?case
  proof (cases rule: range_safe_obtains)
    case _: (1 v)
    then show ?thesis using 1 by (auto simp add: case_memory_def)
  next
  case (2 xs)
  moreover have " $l \in |L$ " by (meson "1.prem1" range_safe_subs)
  ultimately have " $m' \$ l = \text{Some } (\text{mdata.Array } xs)$ " using 1 by auto
  moreover
  have
    "fold ( $\lambda x y. y \gg= (\lambda y'. \text{range\_safe } (\text{fininsert } l \ s) \ m \ x \gg= (\lambda l. \text{Some } (l \cup |y'))$ )) xs (Some
{1|1})
    = fold ( $\lambda x y. y \gg= (\lambda y'. \text{range\_safe } (\text{fininsert } l \ s) \ m' \ x \gg= (\lambda l. \text{Some } (l \cup |y'))$ )) xs (Some
{1|1})"
  proof (rule fold_same)
    show " $\forall x \in \text{set } xs. \text{range\_safe } (\text{fininsert } l \ s) \ m \ x = \text{range\_safe } (\text{fininsert } l \ s) \ m' \ x$ "
    proof
      fix x assume "x  $\in$  set xs"
      moreover from 'x  $\in$  set xs' obtain y
        where "range_safe (fininsert l s) m x = Some y" and "y  $\subseteq$  L"
        using fold_some_subs[OF 2(3)] by blast
      moreover from 'y  $\subseteq$  L' have " $\forall l' \in |y. m' \$ l' = m \$ l'$ " using 1(3) by blast
      ultimately show "range_safe (fininsert l s) m x = range_safe (fininsert l s) m' x"
        using 1(1)[OF 2(1) 'm' \$ l = Some (mdata.Array xs)'] by simp
    qed
  qed
  ultimately show ?thesis using 2 by (auto simp add:case_memory_def)
  qed
  qed

lemma range_safe_same4:
  assumes "range_safe s m l = Some L"
    and " $\forall l' \in |L. (\exists xs. m' \$ l' = \text{Some } (\text{mdata.Array } xs) \wedge m \$ l' = \text{Some } (\text{mdata.Array } xs)) \vee (\exists xs. m' \$ l' = \text{Some } (\text{mdata.Value } xs))$ "
  shows " $\exists L'. \text{range\_safe } s \ m' \ l = \text{Some } L'$ "
  using assms
proof (induction l arbitrary: L rule:range_safe.induct)
  case (1 s m' l)
  from 1(2) show ?case
  proof (cases rule: range_safe_obtains)
    case _: (1 v)
    then show ?thesis using 1 by (auto simp add: case_memory_def)
  next
  case (2 xs)
  moreover have " $l \in |L$ " by (meson "1.prem1" range_safe_subs)
  ultimately consider "m' \$ l = Some (mdata.Array xs)" | " $\exists xs. m' \$ l = \text{Some } (\text{mdata.Value } xs)$ " using
1 by auto
  then show ?thesis
  proof (cases)
    case a: 1

```

```

    moreover have "fold (λx y. y ≫ (λy'. range_safe (finsert l s) m' x ≫ (λl. Some (l |∪|
y')))) xs (Some {l|}) ≠ None"
  proof (rule fold_f_set_some)
    show "∀a∈set xs. range_safe (finsert l s) m' a ≠ None"
  proof
    fix a assume "a∈set xs"
    moreover from 1(2) obtain L' where "range_safe (finsert l s) m a = Some L'" and "L' |⊆|
L"
      by (meson "2"(2) calculation range_safe_obtains_subset)
    moreover from 1(3) have "∀l'∈|L'. (∃xs. m' $ l' = Some (mdata.Array xs) ∧ m $ l' = Some
(mdata.Array xs)) ∨ (∃xs. m' $ l' = Some (mdata.Value xs))"
      using calculation(3) by auto
    ultimately show "range_safe (finsert l s) m' a ≠ None" using 1(1)[OF 2(1),of xs a _ _ L'] a
  by blast
  qed
  qed
  ultimately show ?thesis using 1 by (auto simp add:case_memory_def)
next
  case _: 2
  then show ?thesis using 1 2 <l |∈| L> by (simp add:case_memory_def nth_safe_def split:
if_split_asm)
  qed
  qed
qed

```

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"
  using assms
proof (induction l arbitrary: L rule:range_safe.induct)
  case (1 s m' l)
  from 1(2) show ?case
  proof (cases rule: range_safe_obtains)
    case _: (1 v)
    then show ?thesis using 1 by (auto simp add: case_memory_def)
  next
    case (2 xs)
    moreover have "l |∈| L" by (meson "1.prem"(1) range_safe_subs)
    ultimately consider "m' $ l = Some (mdata.Array xs)" | "∃xs. m' $ l = Some (mdata.Value xs)" using
1 by auto
    then show ?thesis
  proof (cases)
    case a: 1
    moreover from range_safe_same4[OF 1(2,3)] obtain L' where "range_safe s m' l = Some L'" by
blast
    then have
      "fold (λx y. y ≫ (λy'. range_safe (finsert l s) m' x ≫ (λl. Some (l |∪| y')))) xs (Some
{l|}) = Some L'"
      using 1 a 2 by (auto simp add:case_memory_def)
    then obtain L' where *: "fold (λx y. y ≫ (λy'. range_safe (finsert l s) m' x ≫ (λl. Some (l
|∪| y')))) xs (Some {l|}) = Some (L')" by auto
    moreover have "fset L' ⊆ fset L"
  proof (rule fold_subs[OF _ *])
    from <l |∈| L> show "fset {l|} ⊆ fset L" by auto
  next
    show "∀x∈set xs. ∀L'. range_safe (finsert l s) m' x = Some L' → fset L' ⊆ fset L"
  proof (rule,rule,rule)
    fix x L' assume "x ∈ set xs"
    and "range_safe (finsert l s) m' x = Some L'"
    moreover obtain L'' where "range_safe (finsert l s) m x = Some L''" and "L'' |⊆| L"
      using "1.prem"(1) "2"(2) calculation(1) range_safe_obtains_subset by blast
    moreover have "∀l'∈|L''. (∃xs. m' $ l' = Some (mdata.Array xs) ∧ m $ l' = Some

```

```

(mdata.Array xs)  $\vee$  ( $\exists$ xs. m' $ l' = Some (mdata.Value xs))" using 1(3) <L'' | $\subseteq$ | L> by auto
  ultimately show "fset L'  $\subseteq$  fset L" using 1(1)[OF 2(1) a, of x _ _ L''] by fastforce
  qed
  qed
  ultimately show ?thesis using 1 a by (auto simp add:case_memory_def)
next
  case _: 2
  then show ?thesis using 1 2 <l | $\in$ | L> by (simp add:case_memory_def nth_safe_def split:
if_split_asm)
  qed
qed
qed

lemma range_safe_subset_same:
  assumes "range_safe s m l = Some x"
  and "s' | $\subseteq$ | s"
  shows "range_safe s' m l = Some x"
  using assms
proof (induction arbitrary: s x rule:range_safe.induct)
  case (1 s' m l)
  from 1(2) show ?case
  proof (cases rule: range_safe_obtains)
    case _: (1 v)
    then show ?thesis using 1 by (auto simp add: case_memory_def)
  next
    case (2 xs)
    moreover
    have
      "fold ( $\lambda$ x y. y  $\gg$  ( $\lambda$ y'. range_safe (finsert l s) m x  $\gg$  ( $\lambda$ l. Some (l | $\cup$ | y')))) xs (Some
{1|1})
      = fold ( $\lambda$ x y. y  $\gg$  ( $\lambda$ y'. range_safe (finsert l s') m x  $\gg$  ( $\lambda$ l. Some (l | $\cup$ | y')))) xs (Some
{1|1})"
    proof (rule fold_same)
      show " $\forall$ x $\in$ set xs. range_safe (finsert l s) m x = range_safe (finsert l s') m x"
      proof
        fix x assume "x  $\in$  set xs"
        moreover from 'x  $\in$  set xs' obtain y
          where "range_safe (finsert l s) m x = Some y" using fold_some_subst[OF 2(3)] by blast
        ultimately show "range_safe (finsert l s) m x = range_safe (finsert l s') m x" using 1 2
          by (metis fin_mono finsert_mono)
      qed
    qed
    ultimately show ?thesis using 1(3) by (auto simp add:case_memory_def)
  qed
qed

lemma range_safe_in_subst:
  assumes "range_safe s m l = Some L"
  and "l' | $\in$ | L"
  shows " $\exists$ L'. range_safe s m l' = Some L'  $\wedge$  L' | $\subseteq$ | L"
  using assms
proof (induction l arbitrary: L rule:range_safe.induct[where ?a0.0=s])
  case (1 s m l)
  from 1(2) show ?case
  proof (cases rule:range_safe_obtains)
    case _: (1 v)
    then show ?thesis using 1(3) apply auto
      using "1.premis"(1) by auto
  next
    case (2 xs)
    show ?thesis
    proof (cases "l = l'")
      case True
      then have "range_safe s m l' = Some L"

```

```

    using 2 1 by (auto simp add:case_memory_def)
    then show ?thesis using fold_some[OF 2(3)] by simp
  next
    case False
    then obtain x L'' where "x ∈ set xs" and "range_safe (finsert l s) m x = Some L''" and "l' |∈| L''" and "L'' |⊆| L" using 1(3) fold_some_ex[OF 2(3)] fold_some_subs[OF 2(3)] by fastforce
    then have "∃L'. range_safe s m l' = Some L' ∧ L' |⊆| L''" using 1(1)[OF 2(1,2)]
      by (meson range_safe_subset_same fsubset_finsertI)
    then show ?thesis using 'L'' |⊆| L' by auto
  qed
qed
qed

```

lemma range_safe_disj:

```

"∀L. range_safe s m l = Some L → s |∩| L = {||}"(is "?P s m l")
proof (induction rule: range_safe.induct[where ?P = ?P])
  case (1 s m l)
  show ?case
  proof (rule allI, rule impI)
    fix L
    assume "range_safe s m l = Some L "
    then show "s |∩| L = {||}"
    proof (cases rule: range_safe_obtains)
      case (1 v)
      then show ?thesis by simp
    next
      case (2 xs)
      moreover from 2 have
        *: "∀x ∈ set xs. ∀L. range_safe (finsert l s) m x = Some L → finsert l s |∩| L = {||}"
        using 1(1) by simp
      ultimately show ?thesis using fold_disj[of xs "range_safe (finsert l s) m" s _ L] by blast
    qed
  qed
qed

```

lemma range_range:

```

assumes "range_safe s0 m l1 = Some L1"
    and "range_safe s1 m l1 = Some L2"
  shows "L1 = L2"
using assms
by (metis inf_sup_ord(1) range_safe_disj range_safe_subset_same option.inject)

```

lemma range_safe_prefix:

```

assumes "prefix m m'"
    and "range_safe s m l = Some L"
  shows "range_safe s m' l = Some L"
using assms
proof (induction s m' l arbitrary: L rule: range_safe.induct)
  case (1 s' m' l)

  from 1(3) show ?case
  proof (cases rule: range_safe_obtains)
    case *: (1 v)
    then have "m' $ l = Some (mdata.Value v)" using 1(2) unfolding prefix_def nth_safe_def
      by (simp split:if_split_asm add: nth_append)
    then show ?thesis using * by (simp add:case_memory_def)
  next
    case (2 xs)
    then have m'_1: "m' $ l = Some (mdata.Array xs)" using 1(2) unfolding prefix_def nth_safe_def
      by (simp split:if_split_asm add: nth_append)
    moreover have
      "fold (λx y. y ≍ (λy'. range_safe (finsert l s') m x ≍ (λl. Some (l |∪| y')))) xs (Some {||})
      = fold (λx y. y ≍ (λy'. range_safe (finsert l s') m' x ≍ (λl. Some (l |∪| y')))) xs (Some

```

```

{|l|})"
  (is "fold (?P m) xs (Some {|l|}) = fold (?P m') xs (Some {|l|})")
proof (rule take_all1)
  show "∀n≤length xs. fold (?P m) (take n xs) (Some {|l|}) = fold (?P m') (take n xs) (Some
{|l|})"
proof (rule allI, rule impI)
  fix n
  assume "n ≤ length xs"
  then show "fold (?P m) (take n xs) (Some {|l|}) = fold (?P m') (take n xs) (Some {|l|})"
proof (induction n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc(2) have "n < length xs" by auto
  moreover from 2(3) obtain x
  where *: "fold (?P m) (take n xs) (Some {|l|}) ≍
(λy'. range_safe (finsert l s') m (xs ! n) ≍ (λl. Some (l |∪| y')))) = Some x"
  using fold_some_take_some[OF _ <n < length xs>] by metis
  moreover have "?P m' (xs ! n) (fold (?P m') (take n xs) (Some {|l|})) = Some x"
proof -
  from * obtain x' y' where **: "range_safe (finsert l s') m (xs ! n) = Some x'"
  and ***: "(fold (?P m) (take n xs) (Some {|l|})) = Some y'"
  and ****: "x = (x' |∪| y'"
  apply (cases "range_safe (finsert l s') m (xs ! n)")
  apply (auto simp del: range_safe.simps)
  apply (cases "fold (?P m) (take n xs) (Some {|l|})")
  by (auto simp del: range_safe.simps)
  moreover from 1(1)[OF 2(1) m'_l _ _ 1(2), of "(xs ! n)"]
  have "range_safe (finsert l s') m' (xs ! n) = Some x'"
  and "(fold (?P m') (take n xs) (Some {|l|})) = Some y'"
  using ** *** nth_mem Suc.IH <n < length xs> by auto
  ultimately show ?thesis by simp
qed
ultimately show ?case using
  fold_take[of "n" xs "?P m" "(Some {|l|})"]
  fold_take[of "n" xs "?P m'" "(Some {|l|})"]
  by simp
qed
qed
qed
ultimately show ?thesis using 2 by (simp add:case_memory_def)
qed
qed

lemma range_safe_locations:
  assumes "range_safe s m l = Some L"
  and "locations m xs l = Some L'"
  shows "L' |⊆| L"
  using assms
proof (induction xs arbitrary: l L' L s)
  case Nil
  then show ?case by auto
next
  case (Cons i "is")
  then obtain as i' l' L''
  where "m$l = Some (mdata.Array as)"
  and "to_nat i = Some i'"
  and l2: "as $ i' = Some l'"
  and l3: "locations m is l' = Some L''"
  and l4: "L' = (finsert l L'')"
  using locations_obtain[OF Cons(3)] by blast
  then have *:
    "fold (λx y. y ≍ (λy'. (range_safe (finsert l s) m x) ≍ (λl. Some (l |∪| y')))) as (Some {|l|})"

```

3 Stores and State

```

= Some L"
  using range_safe_obtains[OF Cons(2)] by fastforce
moreover from l2 have "l' ∈ set as" unfolding nth_safe_def by (auto split:if_split_asm)
ultimately obtain L0 where **: "range_safe (finsert l s) m l' = Some L0" and "L0 ⊆ L"
  using fold_some_subs by metis
moreover from * have "l ∈ L" using fold_some[OF *] by simp
moreover have "L'' ⊆ L0" by (rule Cons(1)[OF ** 13])
ultimately show ?case using l4 by simp
qed

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"
  by (smt (verit, del_insts) antisym_conv2 assms(1,2,3,4) range_safe_in_subs range_safe_obtains_subset
range_safe_subs pfsubsetD)

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"
  using assms
proof (induction "xs" arbitrary: l L s rule: list_nonempty_induct)
  case (single x)
  then obtain ms i''
    where "m $ l = Some (mdata.Array ms)"
      and "to_nat x = Some i''"
      and 0: "(l', ys, i) = (l, ms, i'')"
    using marray_lookup_obtain_single by blast
  then have
    *: "fold (λx y. y ≫= (λy'. (range_safe (finsert l s) m x) ≫= (λl. Some (l ∪ y')))) ms (Some
{!l!})
= Some L"
    using range_safe_obtains[OF single(1)] by fastforce
  moreover from single(3) 0 have "l'' ∈ set ms" unfolding nth_safe_def by (auto split:if_split_asm)
  ultimately obtain L0
    where **: "range_safe (finsert l s) m l'' = Some L0"
      and "L0 ⊆ L"
    using fold_some_subs by metis
  moreover have "l'' ∈ L0" using range_safe_subs[OF **] by simp
  ultimately show ?case by auto
next
  case (cons x xs)
  then obtain x' xs'
    where xs_def: "xs = x' # xs'"
      by (meson list.exhaust)
  with cons have *: "marray_lookup m (x # x' # xs') l = Some (l', ys, i)" by simp
  then obtain ms i'' l'''
    where "m $ l = Some (mdata.Array ms)"
      and "vtype_class.to_nat x = Some i''"
      and l3: "ms $ i'' = Some l'''"
      and l4: "marray_lookup m xs l''' = Some (l', ys, i)"
    using marray_lookup_obtain_multi xs_def by blast
  then have
    *: "fold (λx y. y ≫= (λy'. (range_safe (finsert l s) m x) ≫= (λl. Some (l ∪ y')))) ms (Some
{!l!})
= Some L"
    using range_safe_obtains[OF cons(3)] by fastforce
  moreover from cons l3 have "l'' ∈ set ms" unfolding nth_safe_def by (auto split:if_split_asm)
  ultimately obtain L0

```

```

where **: "range_safe (finsert l s) m l'' = Some L0"
  and "L0 |⊆| L"
using fold_some_subs by metis
moreover have "l'' |∈| L0" by (rule cons.IH[OF ** 14 cons(5)])
ultimately show ?case by auto
qed

```

```

lemma range_safe_mlookup:
  assumes "range_safe s m l = Some L"
  and "mlookup m xs l = Some l'"
  shows "l' |∈| L"
proof (cases xs)
  case Nil
  then show ?thesis using assms range_safe_subs
  using mlookup_obtain_empty by blast
next
  case (Cons x xs)
  then obtain a xs' i i'
  where "marray_lookup m (x#xs) l = Some (a, xs', i)"
  and "xs' $ i = Some i'"
  and "l' = i'"
  using mlookup_obtain_nempty1 assms(2) by blast
  then show ?thesis using range_safe_marray_lookup[of "x # xs"]
  by (metis assms(1) list.distinct(1))
qed

```

```

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'"
using assms
proof (induction "is" arbitrary: l L')
  case Nil
  then show ?case
  by (metis fempty_fsubset1 fset_eq_fsubset range_safe_subset_same mlookup_obtain_empty
option.inject)
next
  case (Cons i "is")
  then obtain ls i' l''
  where ls_def: "m$l = Some (mdata.Array ls)"
  and "to_nat i = Some i'"
  and l''_def: "ls $ i' = Some l''"
  and "mlookup m is l'' = Some l'"
  using mlookup_obtain_nempty2[OF Cons(2)] by blast
  moreover obtain L'' where "range_safe s' m l'' = Some L''" and "L'' |⊆| L'"
proof -
  from Cons(4) have
  *: "fold (λx y. y ≧ (λy'. range_safe (finsert l s') m x ≧ (λl. Some (l |∪| y')))) ls (Some
{/l/})
  = Some L'"
  using range_safe_obtains ls_def by fastforce
  then obtain L''
  where "range_safe (finsert l s') m l'' = Some L'' ∧ L'' |⊆| L'"
  using fold_some_subs[OF *] l''_def nth_in_set by metis
  then show ?thesis using range_safe_subset_same[of "finsert l s'"] that[of L''] by blast
qed
ultimately show ?case using Cons(1)[OF _ Cons(3)] by blast
qed

```

```

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"

```

```

using assms
proof (induction "is" arbitrary: l L)
  case Nil
  then have "range_safe s m l = Some L" by simp
  then show ?case
  proof (cases rule: range_safe_obtains)
    case (1 v)
    then show ?thesis using Nil by simp
  next
    case (2 xs)
    then show ?thesis using Nil by simp
  qed
next
case (Cons i "is")
from Cons(2) show ?case
proof (cases rule:mlookup_obtain_nempty2)
  case (1 ls i' l'')
  with Cons(3) have
    "(fold (λx y. y ≫= (λy'. (range_safe (finsert l s) m x) ≫= (λl. Some (l |∪| y')))) ls (Some
{||}))
  = Some L"
  by (simp add:case_memory_def split: if_split_asm)
  moreover have "l'' ∈ set ls" using 1 nth_in_set by fast
  ultimately obtain L' where "range_safe (finsert l s) m l'' = Some L'" and "L' |⊆| L"
  using fold_some_subs[of "range_safe (finsert l s) m" ls "Some {||}" L] using 1 by blast
  then have "range_safe s m l'' = Some L'" unfolding range_safe_prefix
  using range_safe_subset_same by blast
  then show ?thesis using Cons(1)[of l''] 1 by simp
qed
qed

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 (rule ccontr)
  assume "¬ l |∉| L'"

  from assms obtain ls
  where ls_def: "m$l = Some (mdata.Array ls)"
  by (meson locations_obtain mlookup_locations_some)
  then have
    L_def: "fold (λx y. y ≫= (λy'. (range_safe (finsert l s) m x) ≫= (λl. Some (l |∪| y'))))
    ls (Some {||}) = Some L"
  using range_safe_obtains[OF assms(2)] by auto

  from ls_def obtain i' l''
  where l''_def: "ls $ i' = Some l''"
  and "mlookup m is l'' = Some l'"
  using mlookup_obtain_nempty2[OF assms(1)] by (metis mdata.inject(2) option.inject)
  moreover from l''_def have "l'' ∈ set ls" unfolding nth_safe_def by (auto split:if_split_asm)
  then obtain L''
  where L''_def: "range_safe (finsert l s) m l'' = Some L''"
  using fold_some_subs[OF L_def] by blast
  then have "range_safe s m l'' = Some L''"
  using range_safe_subset_same [of "finsert l s" m l'' L'']
  by blast
  ultimately have "L' |⊆| L''" using mlookup_range_safe_subs[OF _ assms(3)] by simp
  moreover have "l |∉| L''" using range_safe_disj L''_def by blast
  ultimately show False using '¬ l |∉| L''' by auto
qed

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

```

```

lemma range_subs:
  assumes "range m l = Some X"
  shows "l |∈| X"
  using assms range_safe_subs range_def by metis

lemma range_subs2:
  assumes "range m l = Some X"
  shows "fset X ⊆ loc m"
  using assms range_def range_safe_subs2 by metis

lemma range_same:
  assumes "range m l = Some L"
    and "∀l' |∈| L. m $ l' = m $ l"
  shows "range m' l = Some L"
  using assms range_def range_safe_same by metis

lemma range_prefix:
  assumes "prefix m m'"
    and "range m l = Some L"
  shows "range m' l = Some L"
  using assms range_safe_prefix
  by (metis range_def)

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"
  using assms
proof (induction xs arbitrary: l L s)
  case Nil
  then show ?case
    by (metis bot.extremum range_def range_safe_subset_same mlookup.simps(1) option.inject order_refl)
next
  case (Cons a xs)
  then obtain ls i' l''
    where *: "m $ l = Some (mdata.Array ls)"
      and **: "to_nat a = Some i'"
      and ***: "ls $ i' = Some l''"
      and "mlookup m xs l'' = Some l'"
    using mlookup_obtain_nempty2 by blast
  moreover from Cons(2) have
    "fold (λx y. y ≫ (λy'. range_safe (finsert l s) m x ≫ (λl. Some (l |∪| y')))) ls (Some {|l|})
    = Some L"
  using range_safe_obtains[OF Cons(2)] * ** *** by auto
  then obtain L''
    where "range_safe (finsert l s) m l'' = Some L''"
      and "L'' |⊆| L"
  using fold_some_subs[of "range_safe (finsert l s) m" ls "(Some {|l|})" L l''] ***
  by (meson nth_in_set)
  ultimately show ?case using Cons(1) by blast
qed

lemma range_locations:
  assumes "range m l = Some L"
    and "locations m xs l = Some L'"
  shows "L' |⊆| L"
  using assms range_safe_locations
  by (metis range_def)

lemma range_safe_in_range:
  assumes "range_safe s m l = Some L"
    and "l' |∈| L"
    and "m $ l' = Some (mdata.Array xs)"

```

3 Stores and State

```

    and "xs $ i = Some i'"
    shows "∃L'. range m i' = Some L' ∧ L' |⊆| L"
using assms
proof (induction arbitrary: L rule:range_safe.induct[where ?a0.0 = s and ?a1.0 = m and ?a2.0 = l])
case (1 s m l)
from 1(2) show ?case
proof (cases rule: range_safe_obtains)
case (1 v)
then show ?thesis
using "1.premis"(2,3) by auto
next
case (2 xs)
then consider
  (eq) "l' = l"
  | (2) i L'
  where "i < length xs"
  and "range_safe (fininsert l s) m (xs ! i) = Some L'"
  and "l' |∈| L'"
  using fold_union_in[of "range_safe (fininsert l s) m" xs "{|l|}" L l'] 1(3)
  by blast
then show ?thesis
proof (cases)
case eq
then have "xs $ i = Some i'" using 1 2 by simp
then have "i' ∈ set xs" using nth_in_set by fast
then obtain A
  where "range_safe (fininsert l s) m i' = Some A"
  and "A |⊆| L"
  using 2(3) fold_some_subs[of "range_safe (fininsert l s) m"] by blast
then show ?thesis
  by (metis range_def range_safe_subset_same fempty_fsubsetI)
next
case 22: 2
have "xs!i ∈ set xs" using 22 by simp
moreover have "L' |⊆| L" using 22(2) fold_some_subs[OF 2(3) 'xs!i ∈ set xs'] by simp
ultimately show ?thesis using 1(1)[OF 2(1,2) _ _ 22(2,3) 1(4,5)] by fast
qed
qed
qed

```

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 -

```

from assms obtain L' where "range m i' = Some L'" using range_safe_in_range by blast
then show ?thesis using assms(5,6)
  by (metis range_prefix)

```

qed

lemma range_mlookup:

```

assumes "range m l = Some L"
and "mlookup m xs l = Some l'"
shows "l' |∈| L"
using assms range_safe_mlookup
  by (metis range_def)

```

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 -
  have
    "∀ l' |∈| the (range_safe {||} m l).
      (∃ xs. m' $ l' = Some (mdata.Array xs) ∧ m $ l' = Some (mdata.Array xs))
      ∨ (∃ xs. m' $ l' = Some (mdata.Value xs))"
  proof rule
    fix l'' assume *: "l'' |∈| the (range_safe {||} m l)"
    show
      "(∃ xs. m' $ l'' = Some (mdata.Array xs) ∧ m $ l'' = Some (mdata.Array xs))
      ∨ (∃ xs. m' $ l'' = Some (mdata.Value xs))"
    proof (cases "l''=l'")
      case True
      then show ?thesis using assms by (simp add:nth_safe_def)
    next
      case False
      then show ?thesis
        using assms(1,2) range_safe_sub2[of "{||}" m l "(the (range_safe {||} m l))"] *
        by (auto intro: mdata.exhaust simp add:nth_safe_def range_def loc_def)
    qed
  qed
then show ?thesis
  using assms(1) range_safe_sub3[of "{||}" m l "(the (range_safe {||} m l))" m']
  by (simp add: range_def)
qed

```

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
        (λ v. Some (Value v))
        (λ xs. those (map (read_safe (finsert l s) m) xs) ≧≧ Some o Array))"
by pat_completeness auto
termination
  apply (relation "measure (λ(s,m,_). card ({0..length m} - fset s))")
  using card_less_card by auto

```

```

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"
  using assms
  apply (cases "m $ l", auto split:if_split_asm mdata.split_asm simp add:case_memory_def)
  by (case_tac "those (map (read_safe (finsert l s) m) x2)") auto

```

```

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"

```

```

using assms
apply (auto simp add:case_memory_def split:if_split_asm)
apply (case_tac "those (map (read_safe (finsert l s) m0) ls)",auto)
by (metis (no_types, lifting) bind.bind_lunit cd'_def map_eq_imp_length_eq those_map_nth
those_some_map)

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 -
  {fix s
  have
    "∀cd m'. read_safe s m l = Some cd ⟶ m' = m[l' := mdata.Value v]
    ⟶ (∃cd'. read_safe s m' l = Some cd'" (is "?P s m l")
  proof (induction rule:read_safe.induct[where ?P="?P"])
  case i: (l s m' l)
  show ?case
  proof (rule, rule, rule, rule)
  fix cd m''
  assume 1: "read_safe s m' l = Some cd"
  and 2: "m'' = m'[l' := mdata.Value v]"
  from 1
  show "∃cd'. read_safe s m'' l = Some cd'"
  proof (cases rule: read_safe_cases)
  case (basic v)
  then show ?thesis
  proof (cases "l=l'")
  case True
  then show ?thesis using 1 2
  by (auto simp add: case_memory_def nth_safe_def split:if_split_asm)
  next
  case False
  then have "m'' $ l = Some (mdata.Value v)"
  using basic(1) 2 unfolding nth_safe_def by (auto split:if_split_asm)
  then show ?thesis using 1
  by (auto simp add: case_memory_def)
  qed
  qed
  next
  case (array xs as)
  then show ?thesis
  proof (cases "l=l'")
  case True
  then show ?thesis using 1 2
  by (auto simp add: case_memory_def nth_safe_def split:if_split_asm)
  next
  case False
  then have "m'' $ l = Some (mdata.Array xs)"
  using array(2) 2 unfolding nth_safe_def by (auto split:if_split_asm)
  moreover have "∃as. those (map (read_safe (finsert l s) m'') xs) = Some as"
  proof -
  have "∀x cd. x ∈ set xs ∧
  read_safe (finsert l s) m' x = Some cd ⟶
  (∃cd'. read_safe (finsert l s) m'' x = Some cd'"
  using i[OF array(1,2)] 2 by blast
  then have "∀x. x ∈ set xs ⟶ (∃cd'. read_safe (finsert l s) m'' x = Some cd'"
  using those_map_none[OF array(3)] by simp
  then show ?thesis
  using those_map_some_some[of xs "read_safe (finsert l s) m''"] by auto
  qed
  qed
  ultimately show ?thesis using array(1)
  by (auto simp add: case_memory_def)
  qed
  qed
  qed
  }

```

```

qed
qed}
then show ?thesis using assms by metis
qed

```

```

lemma read_safe_subset_same:
  assumes "read_safe s m l = Some x"
    and "s' |C| s"
  shows "read_safe s' m l = Some x"
  using assms
proof (induction arbitrary: s x rule:read_safe.induct)
  case (1 s' m l)
  show ?case
    apply (auto split:if_split_asm option.split_asm mdata.split_asm simp add: case_memory_def)
    using 1 apply auto[1]
    apply (case_tac "m$1") apply auto
    using 1 apply (auto split:if_split_asm option.split_asm mdata.split_asm simp add:
case_memory_def)[1]
    apply (case_tac "a") apply auto
    using 1 apply (auto split:if_split_asm option.split_asm mdata.split_asm simp add:
case_memory_def)[1]
    using 1(1)[of _ _ "finsert l s"] 1(2) 1(3)
    apply (auto simp del: read_safe.simps split:if_split_asm option.split_asm mdata.split_asm simp add:
case_memory_def)[1]
    by (smt (verit) "1.IH" bind_eq_Some_conv comp_apply data.read_safe_cases finsert_mono
mdata.distinct(1)
mdata.inject(2) option.inject those_map_eq those_map_none)
qed

```

```

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"
  using assms
proof (induction arbitrary: s2 cd1 cd2 l2 rule:read_safe.induct)
  case (1 s1 m l1)
  from 1(3)
  show ?case
  proof (cases rule: read_safe_cases)
    case basic1: (basic v1)
    from 1(4)
    show ?thesis
  proof (cases rule: read_safe_cases)
    case _: (basic v2)
    then show ?thesis using basic1(1,3) 1(2) by simp
  next
    case (array xs as)
    then show ?thesis using basic1(1,3) 1(2) by simp
  qed
  qed
next
  case array1: (array xs1 as1)
  then have
    IH:" $\bigwedge x$  cd1 s2 l2 cd2.
    x  $\in$  set xs1
     $\implies$  read_safe (finsert l1 s1) m x = Some cd1
     $\implies$  read_safe s2 m l2 = Some cd2
     $\implies$  m $ x = m $ l2
     $\implies$  cd1 = cd2"
  using 1(1) by blast
  from 1(4)
  show ?thesis
  proof (cases rule: read_safe_cases)
    case (basic v2)

```

```

then show ?thesis using array1(2) 1(2) by simp
next
case array2: (array xs2 as2)
then have "xs1 = xs2" using 1(2) array1(2) by simp
moreover have
  "those (map (read_safe (finsert l1 s1) m) xs1)
   = those (map (read_safe (finsert l2 s2) m) xs1)"
proof (rule those_map_eq)
show "∀x∈set xs1. ∀y. read_safe (finsert l1 s1) m x = Some y
  → read_safe (finsert l2 s2) m x = Some y"
proof (rule, rule, rule)
  fix x y
  assume "x ∈ set xs1"
  and "read_safe (finsert l1 s1) m x = Some y"
  moreover obtain cd where "read_safe (finsert l2 s2) m x = Some cd"
  using those_map_none[OF array2(3)] <xs1 = xs2> <x ∈ set xs1> by auto
  ultimately show "read_safe (finsert l2 s2) m x = Some y"
  using IH[of x y "(finsert l2 s2)" x] by auto
qed
next
show "∀x∈set xs1. read_safe (finsert l1 s1) m x ≠ None"
  using those_map_none[OF array1(3)] by blast
qed
ultimately have "Array as1 = Array as2" using array1(3) array2(3) by simp
then show ?thesis using array1(4) array2(4) by simp
qed
qed
qed

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 -
  have "∀m' c. prefix m m' ∧ read_safe s m l = Some c
    → read_safe s m' l = Some c" (is "?PROP m s l")
  proof (induction rule:read_safe.induct [where ?P="λs m l. ?PROP m s l"])
  case (1 s m l)
  show ?case
  proof (rule allI, rule allI, rule impI, erule conjE)
  fix m' c
  assume *: "prefix m m'" and **: "read_safe s m l = Some c"
  then have "l < length m" using nth_safe_length[of m l]
  by (auto split:option.split_asm if_split_asm simp add:case_memory_def)
  then have ***: "m'$l = m $ l" using * unfolding prefix_def
  by (metis length_append nth_append nth_safe_some trans_less_add1)

  from **
  consider x
  where "¬ l |∈| s"
  and "m $ l = Some (mdata.Value x)"
  and "c = Value x"
  | xs'
  where "¬ l |∈| s"
  and "m $ l = Some (mdata.Array xs'"
  and "Some c = those (map (read_safe (finsert l s) m) xs') ≧≧ Some o Array"
  using that
  by (auto split:option.split_asm mdata.split_asm if_split_asm simp add:case_memory_def)
  then show "read_safe s m' l = Some c"
  proof cases
  case 1
  then show ?thesis using ***
  by (auto split:option.split mdata.split if_split_asm simp add:case_memory_def)
  next

```

```

case 2
then obtain ar
  where "c = Array ar"
  and "those (map (read_safe (finsert l s) m) xs') = Some ar"
  by (smt (verit, ccfv_SIG) bind_eq_Some_conv comp_apply option.inject)
then have "∀x∈set xs'. read_safe (finsert l s) m x ≠ None"
  using those_map_none by blast
moreover from 1[OF 2(1) 2(2)] have
  IH: "∀x ∈ set xs'. (∀c. read_safe (finsert l s) m x = Some c
  → read_safe (finsert l s) m' x = Some c)"
  using * by blast
ultimately have
  "those (map (read_safe (finsert l s) m) xs')
  = those (map (read_safe (finsert l s) m') xs')"
  using those_map_eq by blast
moreover have "m' $ l = Some (mdata.Array xs')" using *** 2(2) by auto
ultimately show ?thesis using 2 by (auto simp add:case_memory_def)
qed
qed
qed
then show ?thesis using assms by blast
qed

```

```

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"
  using assms by (auto simp add: case_memory_def split:if_split_asm)

```

```

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 -
  from assms obtain ls i' l''
  where *: "m0 $ l1 = Some (mdata.Array ls)"
  and "to_nat i = Some i'"
  and "ls $ i' = Some l''"
  and "mlookup m0 is l'' = Some l1'"
  using mlookup_obtain_empty2 by blast
  moreover from assms * 'to_nat i = Some i'' 'ls $ i' = Some l'''
  obtain cd'
  where cd'_def: "read_safe (finsert l1 s) m0 l'' = Some cd'"
  using those_map_some_nth[of "read_safe (finsert l1 s) m0" ls _ i' l'']
  by (case_tac "those (map (read_safe (finsert l1 s) m0) ls)",
  auto simp add:case_memory_def split:if_split_asm)
  ultimately show ?thesis using that by simp
qed

```

```

definition "read = read_safe {||}"

```

```

lemma read_some_same:
  assumes "read_safe s m l = Some x"
  shows "read m l = Some x"
  using assms read_safe_subset_same unfolding read_def by blast

```

```

lemma read_append:
  assumes "prefix m m'"

```

```

    and "read m l = Some c"
    shows "read m' l = Some c"
    using assms read_safe_prefix unfolding read_def
    by blast

```

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"
  using assms
proof (induction arbitrary: L rule:range_safe.induct)
  case (1 s m l)
  from 1(2) show ?case
  proof (cases rule:range_safe_obtains)
    case (1 v)
    then show ?thesis by (auto simp add:case_memory_def)
  next
    case (2 xs)
    moreover have "∃x. those (map (read_safe (finsert l s) m) xs) ≫= Some ∘ Array = Some x"
    proof -
      from 2(3) have "∀x ∈ set xs. ∃y. range_safe (finsert l s) m x = Some y"
        by (metis fold_some_subs)
      then have "∀x ∈ set xs. ∃y. read_safe (finsert l s) m x = Some y"
        using 1(1)[OF 2(1,2)] by blast
      then obtain z where "those (map (read_safe (finsert l s) m) xs) = Some z"
        by (metis not_Some_eq those_map_none_none)
      then show ?thesis by simp
    qed
  ultimately show ?thesis by (auto simp add:case_memory_def)
qed
qed

```

```

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"
  using assms
proof (induction arbitrary: cd L rule:read_safe.induct)
  case (1 s m' l)
  from 1(2) show ?case
  proof (cases rule:read_safe_cases)
    case _: (basic v)
    then show ?thesis using 1 by (auto simp add:case_memory_def)
  next
    case (array v as)
    moreover have "l ∈ |L|" using 1(3)
    apply (auto simp add:case_memory_def)
    using "1.prem1"(2) data.range_safe_subs by blast
    then have *: "m' $ l = Some (mdata.Array v)" using array(2) 1(4) by simp
    moreover have "those (map (read_safe (finsert l s) m') v)
      = those (map (read_safe (finsert l s) m) v)"
    proof -
      have "∀x ∈ set v. read_safe (finsert l s) m' x = read_safe (finsert l s) m x"
      proof
        fix x assume "x ∈ set v"
        moreover from array(3)
        obtain xx where "those (map (read_safe (finsert l s) m) v) = Some xx"
          by (cases "those (map (read_safe (finsert l s) m) v)", auto)
        then obtain c where "read_safe (finsert l s) m x = Some c"
          by (meson 'x ∈ set v' not_None_eq those_map_none)
        moreover from array have
          **: "fold

```

```

      (λx y. y ≍ (λy'. (range_safe (finsert l s) m x) ≍ (λl. Some (l |∪| y'))))
    v
    (Some {|l|})
  = Some L"
  using range_safe_obtains[OF 1(3)] by auto
  then obtain L' where "range_safe (finsert l s) m x = Some L'" and "L' |⊆| L"
  using fold_some_subs[OF **] 'x ∈ set v' by auto
  moreover from 'L' |⊆| L' have "∀l' |∈| L'. m' $ l' = m $ l'" using 1(4) by blast
  ultimately show "read_safe (finsert l s) m' x = read_safe (finsert l s) m x"
  using 1(1)[OF array(1) *] "1.premis"(3) by auto
qed
then show ?thesis
  by (metis map_ext)
qed
ultimately show ?thesis by (auto simp add:case_memory_def)
qed
qed

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"
  using assms read_safe_range_safe unfolding read_def range_def by blast

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"
  using range_safe_nin_same[OF assms(2,3)] assms(1)
  by (metis read_some_same range_safe_read_safe)

lemma range_read_some:
  assumes "read_safe s m0 l0 = Some cd0"
  shows "∃L. range_safe s m0 l0 = Some L"
  using assms
proof (induction arbitrary: cd0 rule:read_safe.induct)
  case (1 s m l)
  from 1(2) show ?case
  proof (cases rule:read_safe_cases)
    case (basic v)
    then show ?thesis by (auto simp add:case_memory_def)
  next
  case (array xs as)
  moreover have "∀x ∈ set xs. ∃L. range_safe (finsert l s) m x = Some L"
  proof
    fix x
    assume "x ∈ set xs"
    moreover obtain cd where "read_safe (finsert l s) m x = Some cd"
      by (meson array(3) calculation set_nth_some those_map_some_nth)
    ultimately show "∃L. range_safe (finsert l s) m x = Some L" using 1(1)
      by (meson array(1,2))
  qed
  ultimately have
    "fold
      (λx y. y ≍ (λy'. range_safe (finsert l s) m x) ≍ (λl. Some (l |∪| y'))))
      xs
      (Some {|l|})
    ≠ None"
  using fold_f_set_some[of _ "range_safe (finsert l s) m"] by simp
  then show ?thesis using array by (simp add: case_memory_def)
qed
qed

```

```

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 -
  from assms(3,4) have "l1' |∈| L1" using range_safe_mlookup by blast
  then obtain L1'
  where *: "range_safe s m l1' = Some L1'"
    and "L1' |⊆| L1"
    using range_safe_in_subs[OF assms(4), of l1'] by auto
  moreover from * have
  "fold
    (λx y. y ≫ (λy'. (range_safe (fininsert l1' s) m x) ≫ (λl. Some (l |∪| y'))))
    ls
    (Some {|l1'|})
  = Some L1'"
  using assms(1,2) by (auto simp add:case_memory_def split:if_split_asm)
  then obtain LL
  where "range_safe (fininsert l1' s) m l2 = Some LL"
    and "LL |⊆| L1'"
  using fold_some_subs[of "range_safe (fininsert l1' s) m" ls "Some {|l1'|}" L1'] assms(2) by blast
  then have "range_safe s m l2 = Some LL"
  using range_safe_subset_same by blast
  ultimately show ?thesis using range_safe_read_safe[of s m l2 LL] 'LL |⊆| L1'' by auto
qed

```

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'"
  using assms unfolding disjoint_def by blast

```

```

lemma disjoint_disjoint:
  assumes "disjoint m L"
    and "range m l = Some L"
    and "∀ l |∈| L. m $ l = m' $ l"
  shows "disjoint m' L"
  unfolding disjoint_def
proof (rule,rule,rule,rule,rule,rule,rule,rule,rule,rule)
  fix x xs i j i' j' La L'
  assume *: "x |∈| L"
  and **: "m' $ x = Some (mdata.Array xs)"
  and ***: "i ≠ j
    ∧ xs $ i = Some i'
    ∧ xs $ j = Some j'
    ∧ range m' i' = Some La
    ∧ range m' j' = Some L'"
  moreover from * ** have "m $ x = Some (mdata.Array xs)" using assms(3) by auto
  moreover from assms(2) have "La |⊆| L" using range_safe_in_range[OF _ ***, of " {| } " l i i']
  unfolding range_def using ***
  by (metis assms(3) range_def range_same option.inject)

```

```

with *** have "range m i' = Some La" using range_same[of m' i' La m]
  using assms(3) by auto
moreover from assms(2) have "L' |⊆| L" using range_safe_in_range[OF _ * **, of "{}" l j j']
  unfolding range_def using ***
  by (metis assms(3) range_def range_same option.inject)
with *** have "range m j' = Some L'" using range_same[of m' j' L' m]
  using assms(3) by auto
ultimately show "La |∩| L' = {}" using assms(1) unfolding disjoint_def by blast
qed

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"
  unfolding disjoint_def
proof (rule,rule,rule,rule,rule,rule,rule,rule,rule,rule)
  fix x xs i j i' j' La L'
  assume *: "x |∈| L"
  and **: "m' $ x = Some (mdata.Array xs)"
  and ***: "i ≠ j
    ∧ xs $ i = Some i'
    ∧ xs $ j = Some j'
    ∧ range m' i' = Some La
    ∧ range m' j' = Some L'"
  moreover from * ** assms(1,2) have "m $ x = Some (mdata.Array xs)"
    unfolding prefix_def nth_safe_def loc_def by (auto split: if_split_asm simp add: nth_append_left)
  moreover have "La |⊆| L" using range_safe_in_range[OF assms(4) * **] *** by fastforce
  then have "∀ l' |∈| La. m $ l' = m' $ l'"
    using assms(1,2) unfolding prefix_def loc_def nth_safe_def by (auto simp add: nth_append_left)
  then have "range m i' = Some La" using range_same[of m' i' La m] using calculation(3) by auto
  moreover have "L' |⊆| L" using range_safe_in_range[OF assms(4) * **] *** by fastforce
  then have "∀ l' |∈| L'. m $ l' = m' $ l'" using assms(1,2)
    unfolding prefix_def loc_def nth_safe_def by (auto simp add: nth_append_left)
  then have "range m j' = Some L'" using range_same[of m' j' L' m] using calculation(3) by auto
  moreover have "range m j' = Some L'" using range_same[of m' j' L' m] using calculation(6) by auto
  ultimately show "La |∩| L' = {}" using assms(3) unfolding disjoint_def by (meson <x |∈| L>)
qed

```

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 (induction rule: read_safe.induct[where ?P = ?P])
  case IH: (1 s m1 l1)
  show ?case
  proof (rule allI, rule allI, rule allI, rule allI, rule impI, (erule conjE)+)
    fix is1 L1 cd1 L3
    assume 1: "mlookup m0 is1 l1 = Some l1'"
      and 3: "m1 $ l1' = m0 $ l2"
      and 4: "range_safe s m0 l1 = Some L1"

```

```

and 5: "range_safe s m0 l1' = Some L1'"
and 6: "range_safe s2 m0 l2 = Some L2"
and 7: "∀ l | ∈ | L1 | - | L1'. m1 $ l = m0 $ l"
and 8: "∀ l | ∈ | L2. m1 $ l = m0 $ l"
and 9: "read_safe s m0 l1 = Some cd1"
and 10: "read_safe s2 m0 l2 = Some cd2"
and 11: "s | ∩ | L2 = { | | }"
and 12: "locations m0 is1 l1 = Some L3"
and 13: "L3 | ∩ | L2 = { | | }"
and 14: "l1' | ∉ | L2"
and 15: "disjoint m0 L1"
from 9 have "l1 | ∉ | s" by auto
show "∃ x. read_safe s m1 l1 = Some x"
proof (cases "m1$l1")
  case None
  show ?thesis
  proof (cases "is1 = []")
    case True
    then have "m1$l1 = m0$l2" using 1 3 by simp
    then show ?thesis using None
    by (metis "10" read_safe_cases option.discI)
  next
  case False
  then obtain iv is1' where "is1=iv#is1'"
    using list.exhaust by auto

  have "l1 | ∈ | L1" using 4 using range_safe_subs by blast
  moreover have "l1 | ∉ | L1'"
    using 1 'is1=iv#is1'' noloops[of m0 iv is1' l1 l1' s L1 L1'] 4 5 by simp
  ultimately have "m1$l1 = m0$l2" using 7 by blast
  then show ?thesis using 1 None mlookup_start_some 'is1=iv#is1'' by fastforce
qed
next
case (Some a)
then show ?thesis
proof (cases a)
  case (Value v)
  then show ?thesis using Some 'l1 | ∉ | s' by (simp add:case_memory_def)
next
case (Array ls)
have *: "∀ l ∈ set ls. ∃ x'. read_safe (finsert l1 s) m1 l = Some x'"
proof
  fix l assume "l ∈ set ls"
  then obtain i where l_def: "ls $ i = Some l" using set_nth_some by fast
  then show "∃ x'. read_safe (finsert l1 s) m1 l = Some x'"
  proof (cases "is1 = []")
    case True
    then have "m1$l1 = m0$l2" using 1 3 by simp
    then have "m0$l2 = Some (mdata.Array ls)" using Some Array by simp
    then obtain x y
      where "read_safe s2 m0 l = Some x"
        and "range_safe s2 m0 l = Some y"
        and "y | ⊆ | L2"
    using <l ∈ set ls> 6 10 read_safe_range_safe_subs mlookup.simps(1) by blast
    then have "read_safe s2 m1 l = Some x" and "range_safe s2 m1 l = Some y"
      using 8 read_safe_range_safe[of s2 m0 l x y m1]
        range_safe_same[of s2 m0 l y m1] by blast+
    moreover have "l1 | ∉ | L2" using 14 1 True by simp
    ultimately have "read_safe (finsert l1 s) m1 l = Some x"
      using 11 'y | ⊆ | L2' read_safe_range_safe_same[where ?s' = "finsert l1 s"]
      by blast
    then show ?thesis by simp
  next
  case False

```

```

then obtain iv is1' where "is1=iv#is1'"
  using list.exhaust by auto

from 12 have "l1 |∈| L3" using 'is1=iv#is1'' locations_l_in_L by blast
with 13 have "l1 |∉| L2" by blast

have "l1 |∈| L1" using 4 using range_safe_subs by blast
moreover have "l1 |∉| L1'"
  using 1 'is1=iv#is1'' noloops[of m0 iv is1' l1 l1' s L1 L1'] 4 5 by simp
ultimately have "m1$l1 = m0$l1" using 7 by blast

show ?thesis
proof (cases "to_nat iv = Some i")
  case True
  then have "mlookup m0 is1' l = Some l1'"
    using 1 Some Array l_def 'is1=iv#is1'' 'm1$l1 = m0$l1'
    by (cases is1', auto simp add:case_memory_def)
  moreover from 4
  have
    "fold
      (λx y. y ⋈ (λy'. (range_safe (finsert l1 s) m0 x) ⋈ (λl. Some (l |∪| y'))))
      ls
      (Some {|l1|})
    = Some L1"
    using Some Array 'm1$l1 = m0$l1' 'l1 |∉| s' by (auto simp add:case_memory_def)
  then obtain L1''
    where "range_safe (finsert l1 s) m0 l = Some L1''"
    and "L1'' |⊆| L1"
    using fold_some_subs[of "range_safe (finsert l1 s) m0" ls "Some {|l1|}" L1] 'l ∈ set
ls'
    by blast
  moreover from 9 obtain cd1' where "read_safe (finsert l1 s) m0 l = Some cd1'"
    using Some Array 'm1$l1 = m0$l1' 'l1 |∉| s'
    apply (auto simp add:case_memory_def)
    using those_map_none[of "read_safe (finsert l1 s) m0"] 'l ∈ set ls' by force
  moreover have "range_safe (finsert l1 s) m0 l1' = Some L1'" using 5 <l1 |∉| L1'>
    by (smt (verit, best) finsertE fminusD1 fminusD2 range_safe_nin_same)
  moreover have "l1 |∉| s" using 9 by auto
  moreover from 11 have "finsert l1 s |∩| L2 = {|}" using 'l1 |∉| L2' by simp
  moreover from 12 obtain L3'
    where "locations m0 is1' l = Some L3'"
    and "L3 = finsert l1 L3'"
    using Some Array 'is1=iv#is1'' 'm1$l1 = m0$l1' l_def True
    apply (cases "to_nat iv", auto simp add:case_memory_def)
    by (case_tac "locations m0 is1' l", auto)
  moreover from 7 have "∀l|∈|L1'' |∉| L1'. m1 $ l = m0 $ l"
    using 'L1'' |⊆| L1' by blast
  moreover from 13 have "L3' |∩| L2 = {|}" using 'L3 = finsert l1 L3'' by auto
  moreover from 15 have "disjoint m0 L1''" using 'L1'' |⊆| L1' by blast
  moreover have "l1 |∉| L1'"
    by (metis calculation(2) finsert_not_fempty finter_finsert_left range_safe_disj)
  ultimately show ?thesis using IH[DF _ _ 'l ∈ set ls'] Some Array 3 6 8 10 14 by auto
next
  case False
  then obtain i' l'
    where "to_nat iv = Some i'"
    and "i' ≠ i"
    and "ls$i' = Some l'"
    by (metis "12" Array Some <is1 = iv # is1'> <m1 $ l1 = m0 $ l1>
      locations_obtain mdata.inject(2) option.inject)
  moreover have "∃L. range m0 l = Some L"
  proof -
    have "l1 |∉| s" using 9 by auto
    then have

```

```

"range_safe s m0 l1
= fold
  (λx y. y ≫ (λy'. (range_safe (finsert l1 s) m0 x) ≫ (λl. Some (l |∪| y'))))
  ls
  (Some {|l1|})"
using Some Array 'm1$l1 = m0$l1' by (auto simp add:case_memory_def)
with 4 obtain L where "range_safe (finsert l1 s) m0 l = Some L"
using fold_f_set_none[OF 'l ∈ set ls', of "range_safe (finsert l1 s) m0"]
by fastforce
then show ?thesis unfolding range_def using range_safe_subset_same by blast
qed
then obtain L where L_def: "range m0 l = Some L" by blast
moreover have "∃L'. range_safe (finsert l1 s) m0 l' = Some L'"
proof -
  have "l1 ∉ s" using 9 by auto
  then have
    "range_safe s m0 l1
    = fold
      (λx y. y ≫ (λy'. (range_safe (finsert l1 s) m0 x) ≫ (λl. Some (l |∪| y'))))
      ls
      (Some {|l1|})"
    using Some Array 'm1$l1 = m0$l1' by (auto simp add:case_memory_def)
    moreover have "l' ∈ set ls" using 'ls$i' = Some l'' nth_in_set by fast
    ultimately obtain L where "range_safe (finsert l1 s) m0 l' = Some L"
    using fold_f_set_none[of l' _ "range_safe (finsert l1 s) m0"] using 4 by fastforce
    then show ?thesis unfolding range_def using range_safe_subset_same by blast
  qed
then obtain L' where "range_safe (finsert l1 s) m0 l' = Some L'" by blast
then have L'_def: "range_safe s m0 l' = Some L'"
  using data.range_safe_subset_same by blast
then have "range m0 l' = Some L'" unfolding range_def
  using range_safe_subset_same by blast
ultimately have "(L |∩| L' = {|})"
  using 15 Some Array 'm1$l1 = m0$l1' l_def 'l1 |∈| L1' unfolding disjoint_def by metis
moreover from 1 have "mlookup m0 is1' l' = Some l1'"
  using Some Array 'is1=iv#is1' 'm1$l1 = m0$l1' <ls $ i' = Some l'>
  <vtype_class.to_nat iv = Some i'> mlookup_obtain_nempty2 by fastforce
then have "L1' |⊆| L'" using mlookup_range_safe_subs 5 L'_def by blast
ultimately have "L |∩| L1' = {|}" by blast
moreover have "L |⊆| L1'" using L_def 4
proof -
  from 4 have
    "fold
      (λx y. y ≫ (λy'. (range_safe (finsert l1 s) m0 x) ≫ (λl. Some (l |∪| y'))))
      ls
      (Some {|l1|})
    = Some L1"
    using Some Array 'm1$l1 = m0$l1' 'l1 ∉ s' by (auto simp add:case_memory_def)
  then obtain L1''
  where "range_safe (finsert l1 s) m0 l = Some L1''"
  and "L1'' |⊆| L1"
  using fold_some_subs[of "range_safe (finsert l1 s) m0" ls "Some {|l1|}" L1] 'l ∈ set
ls'

  by blast
  then show ?thesis using L_def unfolding range_def
  by (metis bot.extremum data.range_safe_subset_same option.inject)
qed
ultimately have "∀l|∈|L. m0$l = m1$l" using 7 by force
then have "range m1 l = Some L" using L_def
  using range_same by metis
moreover have "(finsert l1 s) |∩| L = {|}"
proof -
  from 4 have
    "fold

```

```

      (λx y. y ≫= (λy'. (range_safe (finsert l1 s) m0 x) ≫= (λl. Some (l |∪| y'))))
    ls
    (Some {|l1|})
  = Some L1"
  using Some Array 'm1$l1 = m0$l1' 'l1 |∉| s' by (auto simp add:case_memory_def)
then obtain L1''
  where L1''_def: "range_safe (finsert l1 s) m0 l = Some L1''"
    and "L1'' |⊆| L1"
  using fold_some_subs[of "range_safe (finsert l1 s) m0" ls "Some {|l1|}" L1] 'l ∈ set
ls'
  by blast
  moreover have "(finsert l1 s) |∩| L1'' = {|}|" using range_safe_disj L1''_def by
blast
  ultimately show ?thesis using L_def unfolding range_def
  by (metis bot.extremum data.range_safe_subset_same option.inject)
qed
  ultimately have "range_safe (finsert l1 s) m1 l = Some L"
  using range_safe_nin_same[of "{|}|" 'l1 |∉| s' 'L |⊆| L1' unfolding range_def by
blast
  then show ?thesis using range_safe_read_safe by blast
  qed
  qed
  qed
  have "l1 |∉| s" using 9 by auto
  then show ?thesis using Some Array
  apply (auto simp add:case_memory_def) using *
  by (smt (verit, del_insts) bind.bind_lunit comp_def not_None_eq those_map_some_some)
  qed
  qed
  qed
  qed

```

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"
  using update_some[of m0 l1 l1' m1 l2 s0 L1' s1 L2 cd2] assms
  unfolding range_def read_def
  by blast

```

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 -
  from assms(3) obtain cd1 where "read_safe s0 m0 l1 = Some cd1"
  using range_safe_read_safe by blast
  moreover from assms(5) obtain cd2 where "read_safe s1 m0 l2 = Some cd2"
  using range_safe_read_safe by blast
  ultimately obtain x where "read_safe s0 m1 l1 = Some x"
  using update_some_obtains_read[OF assms(1,2,3,4,5,6,7) _ _ assms(8, 9,10,11,12)]
  by blast
  then show ?thesis using range_read_some that by blast
qed

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 (rule allI, rule impI)
  fix xs
  assume "m1 $ x = Some (mdata.Array xs)"
  with assms(1,2,4) have "(∀i j i' j' L L'.
    i ≠ j ∧ xs $ i = Some i' ∧ xs $ j = Some j' ∧ range m0 i' = Some L ∧ range m0 j' = Some L'
    → L |∩| L' = {||})" unfolding disjoint_def by auto
  then show "∀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' = {||}"
  using assms(3,5)
  by (metis <m1 $ x = Some (mdata.Array xs)> assms(4) mdata.inject(2) nth_in_set option.inject)
qed

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'"
  using assms
proof (induction is1 arbitrary: L l1 L1)
  case Nil
  then have "l1 = l1'" by simp
  show ?case
  proof (cases "m1 $ l1'")
    case None
    then show ?thesis
    by (metis Nil.prem(9) range_safe_obtains option.distinct(1))
  next
  case (Some a)
  then show ?thesis
  proof (cases a)
    case (Value x1)
    with Nil(9) Some have "L = {||}" by (simp add:case_memory_def split:if_split_asm)
    moreover from Nil(3) have "l1 |∈| L1" using range_safe_subs by blast
    ultimately show ?thesis by auto
  end
end

```

```

next
  case (Array xs)
  from Nil(3) have "l1 |∈| s" by auto
  with Nil(9) have "fold (λx y. y ≫ (λy'. range_safe (finsert l1 s) m1 x ≫ (λl. Some (l |∪|
y')))) xs (Some {l1}) =
    Some L" using Some Array by (simp add:case_memory_def)
  moreover have "∀x∈set xs. ∀L. range_safe (finsert l1 s) m1 x = Some L → fset L ⊆ fset (L1 |∪|
L2)"
  proof (rule ballI, rule allI, rule impI)
    fix x L'
    assume "x ∈ set xs" and "range_safe (finsert l1 s) m1 x = Some L'"
    moreover have "m1 $ l1 = m1 $ l2'"
    proof -
      have "m0 $ l2' = m1 $ l2'"
        by (metis Nil.prem(5,7) range_safe_subs)
      then show ?thesis using Nil(2) <l1 = l1'> by simp
    qed
    moreover have L2'_def: "range_safe s m1 l2' = Some L2'"
      using Nil.prem(5,7) range_safe_same by blast
    ultimately show "fset L' ⊆ fset (L1 |∪| L2)"
      by (metis Array Some range_safe_obtains_subset less_eq_fset.rep_eq range_range
sup.coboundedI2)
    qed
    moreover have "fset {l1} ⊆ fset (L1 |∪| L2)"
      by (metis Nil.prem(3) finsert_absorb finsert_is_funion less_eq_fset.rep_eq range_safe_subs
sup.cobounded1
sup.coboundedI1)
    ultimately show ?thesis using fold_subs by fast
  qed
qed
next
case (Cons i is1')
obtain ls
  where ls_def: "m0 $ l1 = Some (mdata.Array ls)"
  using Cons(2)
  by (cases is1', auto simp add:case_memory_def split:option.split_asm mdata.split_asm)
then have m1_ls: "m1 $ l1 = Some (mdata.Array ls)"
  by (metis Cons.prem(1,3,4,6) range_safe_subs fminus_iff noloops)

have "l1 |∈| L1" using Cons.prem(3) range_safe_subs by blast

obtain i'' where i''_def: "to_nat i = Some i''" and "i'' < length ls"
  using Cons.prem(1) mlookup_obtain_nempty2
  by (metis ls_def mdata.inject(2) nth_safe_length option.inject)
then have "ls $ i'' = Some (ls ! i'')" by simp

have "l1 |∈| s"
  using Cons.prem(3) by force
with Cons(10) m1_ls have "fold (λx y. y ≫ (λy'. range_safe (finsert l1 s) m1 x ≫ (λl. Some (l
|∪| y')))) ls (Some {l1}) = Some L" by (simp add:case_memory_def)
moreover have "∀x∈set ls. ∀L. range_safe (finsert l1 s) m1 x = Some L → fset L ⊆ fset (L1 |∪|
L2)"
proof
  fix x assume "x ∈ set ls"
  then obtain i' where x_def: "ls ! i' = x" and "i' < length ls"
    by (meson in_set_conv_nth)
  then have "ls $ i' = Some x" by simp
  then show "∀L. range_safe (finsert l1 s) m1 x = Some L → fset L ⊆ fset (L1 |∪| L2)"
  proof (cases "i' = i'")
    case True
    show ?thesis
    proof (rule allI, rule impI)
      fix LLL assume "range_safe (finsert l1 s) m1 x = Some LLL"
      then have "range_safe s m1 x = Some LLL"

```

```

    using data.range_safe_subset_same by blast
    moreover from True have "mlookup m0 is1' x = Some l1'" using Cons(2) <ls ! i' = x> <to_nat i
= Some i''>
    by (metis <ls $ i'' = Some (ls ! i'')> ls_def mdata.inject(2) mlookup_obtain_nempty2
option.inject)
    moreover obtain LL where "range_safe s m0 x = Some LL" and "LL |⊆| L1"
    by (meson Cons.prem(3) <x ∈ set ls> range_safe_obtains_subset fsubset_finsertI
range_safe_subset_same ls_def)
    moreover have "∀l|∈|LL |-| L1'. m1 $ l = m0 $ l'"
    using Cons.prem(6) calculation(4) by blast
    moreover have "disjoint m0 LL"
    by (meson Cons.prem(8) calculation(4) disjoint_subs)
    ultimately have "LLL |⊆| LL |∪| L2'" using Cons(1)[OF _ Cons(3) _ Cons(5) Cons(6) _ Cons(8)]
by blast
    then show "fset LLL ⊆ fset (L1 |∪| L2')"
    using <LL |⊆| L1> by blast
qed
next
case False
then obtain LL where "range_safe (finsert l1 s) m0 x = Some LL" and "LL |⊆| L1"
    using Cons.prem(3) <x ∈ set ls> range_safe_obtains_subset ls_def by blast
then have "range m0 x = Some LL"
    by (metis bot.extremum data.range_def data.range_safe_subset_same)
moreover obtain LLL where LL_def: "range_safe (finsert l1 s) m0 (ls ! i'') = Some LLL" and
"LLL |⊆| L1"
    using Cons.prem(3)
    by (meson <ls $ i'' = Some (ls ! i'')> range_safe_obtains_subset ls_def nth_in_set)
then have LLL_def: "range m0 (ls ! i'') = Some LLL"
    by (metis bot.extremum data.range_def data.range_safe_subset_same)
moreover have "LL |∩| LLL = {||}" using Cons(9) unfolding disjoint_def using False i''_def
x_def <l1 |∈| L1> ls_def <ls $ i'' = Some (ls ! i'')> <ls $ i' = Some x>
LL_def LLL_def using calculation(1) by blast
moreover have "L1' |⊆| LLL"
proof -
    have "mlookup m0 is1' (ls ! i'') = Some l1'"
    using Cons.prem(1) <ls $ i'' = Some (ls ! i'')> i''_def ls_def mlookup_obtain_nempty2 by
fastforce
    moreover from LL_def have "range_safe s m0 (ls ! i'') = Some LLL"
    using data.range_safe_subset_same by blast
    ultimately show ?thesis using mlookup_range_safe_subs[OF _ Cons(5)] by blast
qed
ultimately have "LL |∩| L1' = {||}" by auto
with <LL |⊆| L1> have "∀l|∈|LL. m1 $ l = m0 $ l'" using Cons(7) by blast
then have "range_safe (finsert l1 s) m1 x = Some LL" using Cons(10)
    using <range_safe (finsert l1 s) m0 x = Some LL> data.range_safe_same by blast
with <LL |⊆| L1> show ?thesis by auto
qed
qed
moreover from <l1 |∈| L1> have "fset {|l1|} ⊆ fset (L1 |∪| L2')" by simp
ultimately show ?case using fold_subs[of ls "range_safe (finsert l1 s) m1" "fset (L1 |∪| L2')"
"{|l1|}" L]
    by blast
qed
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"
using assms
proof (induction is1 arbitrary: L l1 L1)
  case Nil
  then have "l1 = l1'" by simp
  show ?case
  proof (cases "m1$l1")
    case None
    then show ?thesis
      by (metis Nil.prem1(10) range_safe_obtains option.distinct(1))
  next
  case (Some a)
  then show ?thesis
  proof (cases a)
    case (Value x1)
    with Nil(10) Some have "L = {|l1|}" by (simp add:case_memory_def split:if_split_asm)
    then show ?thesis
      by (simp add: Some Value disjoint_def)
  next
  case (Array xs)
  show ?thesis unfolding disjoint_def
  proof (rule ballI)
    fix x
    assume "x |∈| L"
    moreover have "l1 |∉| s" using Nil by auto
    with Nil(10) have "fold (λx y. y ≫= (λy'. range_safe (finsert l1 s) m1 x ≫= (λl. Some (l |∪|
y')))) xs
      (Some {|l1|}) = Some L" using Some Array by (simp add:case_memory_def)
    ultimately consider "x = l1" | n L'" where "n < length xs" and "range_safe (finsert l1 s) m1 (xs
! n) = Some L'" "x |∈| L'"
    using fold_union_in by fast
    then show "∀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 cases
    case 1
    show ?thesis
    proof (rule allI, rule impI, (rule allI)+, rule impI)
      fix xs' i j i' j' L L'
      assume "m1 $ x = Some (mdata.Array xs)"
      and *: "i ≠ j ∧ xs' $ i = Some i' ∧ xs' $ j = Some j' ∧ range m1 i' = Some L ∧ range m1
j' = Some L'"
      moreover have "range m0 i' = Some L"
      proof -
        obtain L0 where "range_safe (finsert l2' s) m0 i' = Some L0" and "L0 |⊆| L2'" using
Nil(5)
        by (metis "1" <l1 = l1'> assms(2) calculation(1,2) range_safe_obtains_subset
nth_in_set)
      then have "range m0 i' = Some L0"
        by (metis bot.extremum data.range_def data.range_safe_subset_same)
      moreover from <L0 |⊆| L2'> have "∀l |∈| L0. m1 $ l = m0 $ l" using Nil(7) by blast
      ultimately show ?thesis using * range_same by metis
    qed
    moreover have "range m0 j' = Some L'"
    proof -
      obtain L0 where "range_safe (finsert l2' s) m0 j' = Some L0" and "L0 |⊆| L2'" using
Nil(5)
      by (metis "1" <l1 = l1'> assms(2) calculation(1,2) range_safe_obtains_subset
nth_in_set)
      then have "range m0 j' = Some L0"

```

```

    by (metis bot.extremum data.range_def data.range_safe_subset_same)
    moreover from <L0 |⊆| L2'> have "∀l |∈| L0. m1 $ l = m0 $ l" using Nil(7) by blast
    ultimately show ?thesis using * range_same by metis
  qed
  ultimately show "L |∩| L' = {||}" using Some Array Nil(9) unfolding disjoint_def
    by (metis "1" <l1 = l1'> assms(2,5) range_safe_subs)
  qed
next
case 2
moreover from Nil(5) obtain LL where LL_def: "range_safe (finsert l2' s) m0 (xs ! n) = Some
LL" and "LL |⊆| L2'" using Some Array Nil(2) <l1 = l1'>
  by (metis calculation(1) range_safe_obtains_subset nth_mem)
then have "range_safe (finsert l2' s) m1 (xs ! n) = Some LL"
  by (metis assms(7) data.range_safe_same finsert_fsubset mk_disjoint_finsert)
ultimately have "LL = L'" using range_range by blast
with Nil(9) 2 LL_def <LL |⊆| L2'> Some Array
have *: "∀x|∈|L2'. ∀xs. m0 $ x = Some (mdata.Array xs) → (∀i j i' j' L L'.
i ≠ j ∧ xs $ i = Some i' ∧ xs$j = Some j' ∧ range m0 i' = Some L ∧ range m0 j' = Some L'
→ L |∩| L' = {||})" unfolding disjoint_def
  by (metis)
show ?thesis
proof (rule allI, rule impI, (rule allI)+, rule impI)
  fix xs' i j i' j' L L'
  assume 00: "m1 $ x = Some (mdata.Array xs')"
  and **: "i ≠ j ∧ xs' $ i = Some i' ∧ xs' $ j = Some j' ∧ range m1 i' = Some L ∧ range
m1 j' = Some L'"
  moreover have "x |∈|L2'"
    using "2"(3) <LL = L'> <LL |⊆| L2'> by blast
  moreover have "m0 $ x = Some (mdata.Array xs')"
    using "00" assms(7) calculation(3) by auto
  moreover have "range m0 i' = Some L"
  proof -
    obtain L0 where "range_safe (finsert l2' s) m0 i' = Some L0" and "L0 |⊆| L2'" using
Nil(5) Some Array 2 LL_def <LL |⊆| L2'>
      by (smt (verit, ccfv_threshold) "***" <LL = L'> range_safe_l_in_L calculation(1)
fsubset_trans range_safe_in_subs
nth_in_set)
    then have "range m0 i' = Some L0"
      by (metis bot.extremum data.range_def data.range_safe_subset_same)
    moreover from <L0 |⊆| L2'> have "∀l |∈| L0. m1 $ l = m0 $ l" using Nil(7) by blast
    ultimately show ?thesis using ** range_same by metis
  qed
  moreover have "range m0 j' = Some L'"
  proof -
    obtain L0 where "range_safe (finsert l2' s) m0 j' = Some L0" and "L0 |⊆| L2'" using
Nil(5) Some Array 2 LL_def <LL |⊆| L2'>
      by (smt (verit, ccfv_threshold) "***" <LL = L'> range_safe_l_in_L calculation(1)
fsubset_trans range_safe_in_subs
nth_in_set)
    then have "range m0 j' = Some L0"
      by (metis bot.extremum data.range_def data.range_safe_subset_same)
    moreover from <L0 |⊆| L2'> have "∀l |∈| L0. m1 $ l = m0 $ l" using Nil(7) by blast
    ultimately show ?thesis using ** range_same by metis
  qed
  ultimately show "L |∩| L' = {||}" using * by blast
  qed
qed
qed
qed
qed
next
case (Cons i is1')
obtain ls
  where ls_def: "m0 $ l1 = Some (mdata.Array ls)"

```

```

using Cons(2)
by (cases is1', auto simp add: case_memory_def split: option.split_asm mdata.split_asm)
then have m1_ls: "m1 $ l1 = Some (mdata.Array ls)"
by (metis Cons.prem1s(1,3,4,6) range_safe_subs fminus_iff noloops)

have "l1 ∈ | L1" using Cons.prem1s(3) range_safe_subs by blast

obtain i'' where i''_def: "to_nat i = Some i''" and "i'' < length ls"
using Cons.prem1s(1) mlookup_obtain_nempty2
by (metis ls_def mdata.inject(2) nth_safe_length option.inject)
then have "ls $i'' = Some (ls ! i'')" by simp

show ?case unfolding disjoint_def
proof
fix x
assume "x ∈ | L"
show
"∀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 (rule allI, rule impI)
fix xs
assume xs_def: "m1 $ x = Some (mdata.Array xs)"

have "l1 ∉ | s"
by (metis Cons.prem1s(3) <l1 ∈ | L1> fminus_iff fminus_triv range_safe_disj)
then have
"fold
(λx y. y ≍ (λy'. (range_safe (finsert l1 s) m1 x) ≍ (λl. Some (l |∪| y'))))
ls
(Some {|l1|})
= Some L"
using Cons(11) by (simp add: case_memory_def m1_ls)
with <x ∈ | L> consider
(eq) "x = l1"
| (2) i' L'
where "i' < length ls"
and "range_safe (finsert l1 s) m1 (ls ! i') = Some L'"
and "x ∈ | L'"
using fold_union_in[of "range_safe (finsert l1 s) m1" ls "{|l1|}" L x]
by blast
then show
"∀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 cases
case eq
then have "ls = xs" using xs_def m1_ls by simp

{fix i0 j L0 L' i0' i' j'
assume "i0 = i'"
and "j ≠ i'"
and "i' < length ls"
and *: "i0 ≠ j ∧ xs $ i0 = Some i0' ∧ xs $ j = Some j' ∧ range m1 i0' = Some L0 ∧ range
m1 j' = Some L'"

```

```

have "∀x |∈| L0. x |∉| L'"
proof
  fix x
  assume "x |∈| L0"

  have "mlookup m0 is1' i0' = Some l1'"
    using Cons(2) i''_def ls_def <i0 = i''> <ls $ i'' = Some (ls ! i'')> <ls = xs> *
  by (cases "is1'", auto simp add:case_memory_def)
  moreover obtain Li'
    where Li'_def: "range_safe (finsert l1 s) m0 i0' = Some Li'"
      and "Li' |⊆| L1"
    using range_safe_obtains_subset[OF Cons(4) ls_def] using <i' < length ls> <ls = xs> * by
(metis nth_in_set)
  then have "range_safe s m0 i0' = Some Li'"
    by (meson fsubset_finsertI range_safe_subset_same)
  moreover have "∀l|∈|Li' |-| L1'. m1 $ l = m0 $ l" using Cons(7) <Li' |⊆| L1> by auto
  moreover obtain LL' where "range_safe s m1 i0' = Some LL'" and "LL' |⊆| L"
    using range_safe_obtains_subset[OF Cons(11) m1_ls] <ls = xs> *
    by (metis fsubset_finsertI range_safe_subset_same nth_in_set)
  moreover have "disjoint m0 Li'"
    using Cons.prem8 <Li' |⊆| L1> by auto
  ultimately consider "x |∈|Li'" | "x |∈| L2'"
    using update_some_range_subset[OF _ Cons(3) _ Cons(5,6) _ Cons(8), of is1' "i0'" Li' LL']
    by (metis "*" <x |∈| L0> fsubsetD funion_iff range_def range_range)
  then show "x |∉| L'"
  proof cases
    case 1
    moreover have "L' |∩| Li' = {}"
    proof -
      obtain LL where LL_def: "range m0 j' = Some LL" and "LL |⊆| L1"
        by (metis "*" Cons.prem3 <ls = xs> range_safe_in_range range_safe_subs ls_def)
      moreover have "∀l|∈|LL. m1 $ l = m0 $ l"
      proof -
        have "m0 $ l1 = Some (mdata.Array xs)"
          by (simp add: <ls = xs> ls_def)
        moreover have "range m0 i0' = Some Li'"
          by (metis Li'_def fempty_fsubsetI range_def range_safe_subset_same)
        ultimately have "LL |∩| Li' = {}"
          using <l1 |∈| L1> * LL_def Cons(9) unfolding disjoint_def by blast
        moreover have "L1' |⊆| Li'" using mlookup_range_safe_subs[OF _ Cons(5)]
          using <range_safe s m0 i0' = Some Li''> <mlookup m0 is1' i0' = Some l1''> by blast
        ultimately have "LL |∩| L1' = {}" by auto
        then show ?thesis using Cons(7) <LL |⊆| L1> by auto
      qed
      ultimately have "range m0 j' = Some L'" using range_safe_same[of "{}" m0 j' LL m1] *
        using range_def by argo
    moreover have "m0 $ l1 = Some (mdata.Array xs)"
      by (simp add: <ls = xs> ls_def)
    moreover have "range m0 i0' = Some Li'"
      by (metis Li'_def fempty_fsubsetI range_def range_safe_subset_same)
    ultimately show "L' |∩| Li' = {}"
      using <l1 |∈| L1> * LL_def Cons(9) unfolding disjoint_def by blast
  qed
  ultimately show ?thesis by blast
next
case 2
moreover have "L' |⊆| L1 |-| L1'"
proof -
  obtain LL where LL_def: "range m0 j' = Some LL" and "LL |⊆| L1"
    by (metis "*" Cons.prem3 <ls = xs> range_safe_in_range range_safe_subs ls_def)
  moreover have "LL |∩| L1' = {}"
  proof -
    have "m0 $ l1 = Some (mdata.Array xs)"
      by (simp add: <ls = xs> ls_def)
  qed

```

```

    moreover have "range m0 i0' = Some Li'"
      by (metis Li'_def fempty_fsubsetI range_def range_safe_subset_same)
    ultimately have "LL |∩| Li' = {}"
      using <l1 |∈| L1> * LL_def Cons(9) unfolding disjoint_def by blast
    moreover have "L1' |⊆| Li'" using mlookup_range_safe_subs[OF _ Cons(5)]
      using <range_safe s m0 i0' = Some Li'> <mlookup m0 is1' i0' = Some l1'> by blast
    ultimately have "LL |∩| L1' = {}" by auto
    then show ?thesis using Cons(7) <LL |⊆| L1> by auto
  qed
  ultimately have "∀l|∈|LL. m1 $ l = m0 $ l"
    using Cons.prem(6) by blast
  then have "range m0 j' = Some L'" using range_safe_same[of "{}" m0 j' LL m1] *
    using LL_def range_def by argo
  then show ?thesis
    using LL_def <LL |∩| L1' = {}> <LL |⊆| L1> by auto
  qed
  ultimately show ?thesis using Cons(12) by auto
  qed
  then have "L0 |∩| L' = {}" by auto
} note lem=this

show ?thesis
proof ((rule allI)+, rule impI)
  fix i0 j i0' j' L0 L'
  assume *: "i0 ≠ j ∧ xs $ i0 = Some i0' ∧ xs $ j = Some j' ∧ range m1 i0' = Some L0 ∧ range
m1 j' = Some L'"
  then consider
    (1) "i0 ≠ j" and "i0 ≠ i'" and "j ≠ i'" |
    (2) "i0 = i'" and "j ≠ i'" |
    (3) "j = i'" and "i0 ≠ i'"
  by blast
  then show "L0 |∩| L' = {}"
  proof cases
    case 1
    from *
    have "i0 ≠ j"
      and "xs $ i0 = Some i0'"
      and "xs $ j = Some j'"
      and "range m1 i0' = Some L0"
      and "range m1 j' = Some L'"
    by simp+
    then have "i0' ∈ set xs" and "j' ∈ set xs" by (auto simp add: nth_in_set)
    then obtain Li'
      where Li'_def: "range_safe (finset l1 s) m0 i0' = Some Li'"
      and "Li' |⊆| L1" using range_safe_obtains_subset[OF Cons(4) ls_def, of "i0'"]
    using <ls = xs> by auto
    then have "range m0 i0' = Some Li'"
      unfolding range_def using data.range_safe_subset_same by blast
    moreover obtain Li''
      where Li''_def: "range_safe (finset l1 s) m0 j' = Some Li''"
      and "Li'' |⊆| L1" using range_safe_obtains_subset[OF Cons(4) ls_def, of "j'"]
    using <ls = xs> <j' ∈ set xs> by auto
    then have "range m0 j' = Some Li''"
      unfolding range_def using data.range_safe_subset_same by blast
    moreover have "Li' |∩| Li'' = {}"
      using "*" Cons.prem(8) <l1 |∈| L1> <ls = xs> calculation(1,2) disjoint_def ls_def by
blast

    moreover have "range m1 i0' = range m0 i0'" and "range m1 j' = range m0 j'"
  proof -
    have *: "mlookup m0 is1' (ls ! i') = Some l1'"
      using Cons(2) i''_def ls_def <ls $ i'' = Some (ls ! i')>
      by (cases "is1'", auto simp add: case_memory_def)
    moreover obtain LL where LL_def: "range_safe s m0 (ls ! i') = Some LL"

```

```

and "LL |⊆| L1" using range_safe_obtains_subset[OF Cons(4) ls_def]
  by (metis <ls $ i'' = Some (ls ! i'')> fsubset_finsertI range_safe_subset_same
nth_in_set)
moreover have "L1' |⊆| LL" using mlookup_range_safe_subs[OF * Cons(5)] LL_def by simp
moreover have "LL |∩| Li' = {||}"
proof -
  from LL_def have "range m0 (ls ! i'') = Some LL"
  by (metis fempty_fsubsetI range_def range_safe_subset_same)
  then show ?thesis using Cons(9) <range m0 i0' = Some Li''> unfolding disjoint_def
  using "1"(2) <ls $ i'' = Some (ls ! i'')> <ls = xs> <xs $ i0 = Some i0''> ls_def <l1
|∈| L1> by blast
  qed
  ultimately have "Li' |∩| L1' = {||}" by auto
  then have "∀ l |∈| Li''. m1 $ l = m0 $ l" using Cons(7) <range m0 i0' = Some Li''> <Li'
|⊆| L1> by blast
  then show "range m1 i0' = range m0 i0'"
  by (metis <range m0 i0' = Some Li''> data.range_same)

  from LL_def have "range m0 (ls ! i'') = Some LL"
  by (metis fempty_fsubsetI range_def range_safe_subset_same)
  then have "LL |∩| Li'' = {||}"
  using "1"(3) <ls $ i'' = Some (ls ! i'')> <ls = xs> <xs $ j = Some j''> ls_def <l1 |∈|
L1> Cons(9) <range m0 j' = Some Li''>
  unfolding disjoint_def by blast
  then have "Li'' |∩| L1' = {||}" using <L1' |⊆| LL> by auto
  then have "∀ l |∈| Li''. m1 $ l = m0 $ l"
  using Cons(7) <range m0 j' = Some Li''> <Li'' |⊆| L1> by blast
  then show "range m1 j' = range m0 j'" by (metis <range m0 j' = Some Li''>
data.range_same)
  qed
  ultimately show ?thesis
  by (simp add: <range m1 i0' = Some L0> <range m1 j' = Some L''>)
next
case 2
show ?thesis using lem 2
  using "*" <i'' < length ls> by blast
next
case 3
show ?thesis using lem 3
  using "*" <i'' < length ls> by blast
qed
qed
next
case 2
show ?thesis
proof (cases "i'' = i'")
case True
with Cons(2) have "mlookup m0 is1' (ls ! i'') = Some l1'"
  using Cons(2) i''_def ls_def <ls $ i'' = Some (ls ! i'')>
  by (cases "is1'", auto simp add: case_memory_def)
moreover obtain LL
  where "range_safe (finsert l1 s) m0 (ls ! i'') = Some LL"
  and "LL |⊆| L1"
  using range_safe_obtains_subset[OF Cons(4) ls_def] using "2"(1) True nth_mem by blast
then have "range_safe s m0 (ls ! i'') = Some LL"
  by (meson fsubset_finsertI range_safe_subset_same)
moreover have "∀ l |∈| LL |-| L1'. m1 $ l = m0 $ l" using <LL |⊆| L1> Cons(7) by auto
moreover have "disjoint m0 LL" using <LL |⊆| L1> Cons(9) by auto
moreover obtain LL'
  where "range_safe s m1 (ls ! i'') = Some LL'"
  and "LL' |⊆| L"
  using range_safe_obtains_subset[OF Cons(11) m1_ls]
  by (metis <ls $ i'' = Some (ls ! i'')> fsubset_finsertI range_safe_subset_same nth_in_set)
moreover have "LL |-| L1' |∩| L2' = {||}" using Cons(12)

```

```

    using <LL |⊆| L1> by blast
    ultimately have "disjoint m1 LL'" using Cons(1)[of "ls ! i'", OF _ Cons(3) _ Cons(5,6) _
Cons(8) _ Cons(10)]
    by simp
    then show ?thesis
    using <LL' |⊆| L> "2"(2,3) True <range_safe s m1 (ls ! i'') = Some LL'> disjoint_def
range_range xs_def by blast
next
case False
moreover obtain Li'
  where Li'_def: "range_safe (finsert l1 s) m0 (ls ! i') = Some Li'"
    and "Li' |⊆| L1" using 2(1) range_safe_obtains_subset[OF Cons(4) ls_def, of "ls ! i'"]
  by auto
then have "range m0 (ls ! i') = Some Li'"
  unfolding range_def using data.range_safe_subset_same by blast
moreover obtain Li''
  where Li''_def: "range_safe (finsert l1 s) m0 (ls ! i'') = Some Li'''"
    using i''_def range_safe_obtains_subset[OF Cons(4) ls_def, of "ls ! i''] <i'' < length ls>
by auto
then have "range m0 (ls ! i'') = Some Li'''"
  unfolding range_def using data.range_safe_subset_same by blast
moreover have "ls $i' = Some (ls ! i')" by (simp add: "2"(1))
ultimately have "Li' |∩| Li'' = {}"
  using Cons(9) Li''_def <l1 |∈| L1> ls_def <ls $i'' = Some (ls ! i'')>
  unfolding disjoint_def by blast
moreover have "L1' |⊆| Li'''"
proof -
  have "mlookup m0 is1' (ls ! i'') = Some l1'"
    using Cons(2) i''_def ls_def <ls $ i'' = Some (ls ! i'')>
    by (cases "is1'", auto simp add: case_memory_def)
  moreover from Li''_def have "range_safe s m0 (ls ! i'') = Some Li'''"
    using data.range_safe_subset_same by blast
  ultimately show ?thesis using mlookup_range_safe_subs[OF _ Cons(5)] by simp
qed
ultimately have "Li' |∩| L1' = {}" by blast
then have *: "∀x |∈| Li'. m1 $ x = m0 $ x" using Cons(7) <Li' |⊆| L1> by blast
then have "Li' = L'" using 2(2) range_safe_same[OF Li'_def] by auto
moreover from <Li' = L'> have "x |∈| Li'" using 2(3) by simp
ultimately have "m1 $ x = m0 $ x" using * by simp
moreover have "∀l ∈ set xs. range m1 l = range m0 l"
proof
  fix l assume "l ∈ set xs"
  then have "l |∈| L'" using range_safe_l_in_L[OF 2(2,3) xs_def] by simp
  then obtain X
    where "range_safe (finsert l1 s) m1 l = Some X"
      and "X |⊆| L'"
    using range_safe_in_subs[OF 2(2)] by blast
  moreover from <X |⊆| L'> have "X |⊆| Li'" using <Li' = L'> by simp
  then have "∀x |∈| X. m1 $ x = m0 $ x" using * by auto
  ultimately have "range_safe (finsert l1 s) m1 l = range_safe (finsert l1 s) m0 l"
    using range_safe_same[of "finsert l1 s" m1 l] by auto
  then show "range m1 l = range m0 l"
    by (metis <range_safe (finsert l1 s) m1 l = Some X>
      empty_fminus empty_iff range_def range_safe_nin_same)
qed
moreover have "x |∈| L1" using "2"(3) <Li' |⊆| L1> <Li' = L'> by auto
moreover have "m0 $ x = Some (mdata.Array xs)"
  using <m1 $ x = Some (mdata.Array xs)> calculation(1) by auto
ultimately have
  "∀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' = {||}"
    using disjoint_range_disj[OF Cons(9), of x xs m1] by simp
    then show ?thesis using <m1 $ x = Some (mdata.Array xs)> by blast
  qed
qed
qed
qed
qed
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
.

```

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"
  using assms
  apply (cases s)
  apply (auto)
  using alookup.elims apply blast
  apply (cases "is", auto)
  apply (case_tac "to_nat a", auto)
  by (case_tac "x2$aa", auto)

lemma alookup_append:
  "alookup (xs1@xs2) cd = alookup xs1 cd ≧ alookup xs2"
proof (induction xs1 arbitrary: cd)
  case Nil
  then show ?case by simp
next
  case (Cons a xs1)
  then show ?case
  proof (cases cd)
    case (Value x1)
    then show ?thesis by auto
  next
    case (Array x2)
    then show ?thesis using Cons
  end
end

```

```

    apply (case_tac "to_nat a", auto)
    apply (case_tac "x2$aa")
    by auto
qed
qed

lemma alookup_empty_some:
  shows "alookup [] cd = Some cd"
  by simp

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'"
  using assms
  by simp

proposition alookup_same: "( $\forall$ xs. alookup xs cd1 = alookup xs cd2)  $\equiv$  cd1 = cd2"
proof (induction cd1)
  case (Value x1)
  then show ?case
  proof (induction cd2)
    case (Value x2)
    then show ?case apply (auto)
    proof -
      assume "( $\forall$ xs. alookup xs (adata.Value x1) = alookup xs (adata.Value x2))"
      then show "x1 = x2"
      proof (rule contrapos_pp)
        assume "x1  $\neq$  x2"
        then have "alookup [] (adata.Value x1)  $\neq$  alookup [] (adata.Value x2)" by simp
        then show " $\neg$  ( $\forall$ xs. alookup xs (adata.Value x1) = alookup xs (adata.Value x2))" by blast
      qed
    qed
  next
  case (Array x)
  show ?case apply (auto)
  proof -
    have "alookup [] (adata.Value x1)  $\neq$  alookup [] (adata.Array x)" by simp
    then show " $\exists$ xs. alookup xs (adata.Value x1)  $\neq$  alookup xs (adata.Array x)" by blast
  qed
qed
next
case (Array x1)
then show ?case
proof (induction cd2)
  case (Value x2)
  show ?case apply (auto)
  proof -
    have "alookup [] (adata.Array x1)  $\neq$  alookup [] (adata.Value x2)" by simp
    then show " $\exists$ xs. alookup xs (adata.Array x1)  $\neq$  alookup xs (adata.Value x2)" by blast
  qed
next
case (Array x2)
show ?case apply (auto)
proof -
  assume " $\forall$ xs. alookup xs (adata.Array x1) = alookup xs (adata.Array x2)"
  then show "x1 = x2"
  proof (rule contrapos_pp)
    assume "x1  $\neq$  x2"
    then have "alookup [] (adata.Array x1)  $\neq$  alookup [] (adata.Array x2)" by simp
    then show " $\neg$  ( $\forall$ xs. alookup xs (adata.Array x1) = alookup xs (adata.Array x2))" by blast
  qed
qed

```

```
qed
qed
qed
```

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'"
  using assms
proof (induction xs arbitrary: l cd)
  case Nil
  then show ?case
    using alookup.simps(1) mlookup_obtain_empty by blast
next
  case (Cons a xs)
  from Cons(3) obtain xs' l''
  where *: "m$l = Some (mdata.Array xs')"
    and x1: "(to_nat a) ≧ ($ xs' = Some l'')" and **: "mlookup m xs l'' = Some l'"
  using mlookup_obtain_empty1[OF Cons(3)] apply auto
  apply (case_tac xs)
  using marray_lookup_obtain_single marray_lookup_obtain_multi
  apply (auto simp add: case_memory_def split: option.split_asm mdata.split_asm)
  apply (case_tac "to_nat a") apply auto
  apply (case_tac "x2a $ aaa") by auto

  from Cons(2) *
  have *: "Some cd = those (map (aread_safe (finsert l s) m) xs') ≧ Some o adata.Array"
  using a_data.read_safe_cases[of s m l cd] by fastforce
  then obtain xs''
  where xx1: "those (map (aread_safe (finsert l s) m) xs') = Some xs''" by fastforce
  then have a1: "cd = adata.Array xs''" using * by simp

  moreover obtain cd' where a3: "aread_safe s m l'' = Some cd'"
  by (smt (verit, ccfv_threshold) Option.bind_cong bind.bind_lunit bind_rzero
    a_data.read_safe_subset_same fsubset_finsertI option.discI
    those_map_some_nth x1 xx1)
  moreover have a2: "(to_nat a) ≧ ($ xs'' = Some cd'"
  by (smt (verit, ccfv_SIG) a3 a_data.read_some_same bind_eq_Some_conv
    local.x1 those_map_nth those_map_some_nth xx1)
  ultimately obtain cd''
  where "aread_safe s m l' = Some cd'' ∧ alookup xs cd' = Some cd''"
  using Cons(1) ** by auto
  moreover have "alookup xs cd' = alookup (a # xs) cd"
  by (simp add: a1 a2)
```

```
ultimately show ?case by simp
qed
```

```
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'"
  using assms
proof (induction "is" arbitrary: s l1 cd1)
  case Nil
  then show ?case by simp
next
  case (Cons i is')
  then obtain ls i' l'' cd'
  where *: "m0 $ l1 = Some (mdata.Array ls)"
    and "to_nat i = Some i'"
    and "ls $ i' = Some l''"
```

```

    and "mlookup m0 is' l'' = Some l1'"
    and cd'_def: "aread_safe (finsert l1 s) m0 l'' = Some cd'"
  using a_data.mlookup_read_safe_obtain by metis
then obtain cd''
  where "alookup is' cd' = Some cd''" using Cons(1)[of l''] by blast
moreover from * obtain cs
  where "cd1 = Array cs"
    and "cs $ i' = Some cd'"
  using a_data.read_safe_array Cons * cd'_def <ls $ i' = Some l''> by blast
ultimately show ?case using 'to_nat i = Some i'' 'ls $ i' = Some l''' by auto
qed

```

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 o adata.Array o 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'')"
  using assms
  apply (cases "is", auto)
  apply (cases cd, auto)
  apply (case_tac " vtype_class.to_nat a", auto)
  apply (case_tac "x2 $ aa", auto)
  apply (case_tac " aupdate list v ab")
  by auto

```

```

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''"
  using assms
  apply (cases "as $ i'", auto)
  by (case_tac "aupdate is v a", auto)

```

```

lemma aupdate_alookup:
  assumes "aupdate is v cd = Some cd'"
  shows "alookup is cd' = Some v"
  using assms
proof (induction "is" arbitrary:cd cd')
  case Nil
  then show ?case by auto
next
  case (Cons i "is'")
  then obtain xs i' i'' cd''
  where "cd=adata.Array xs"
    and "to_nat i = Some i'"
    and "xs $ i' = Some i''"
    and *: "aupdate is' v i'' = Some cd''"

```

3 Stores and State

```

    and "cd' = adata.Array (list_update xs i' cd'')"
    using aupdate_obtain[of "i # is'" v cd cd']
    by blast
    moreover from * have "alookup is' cd'' = Some v" using Cons.IH by blast
    ultimately show ?case by (auto simp add: nth_safe_def split:if_split_asm)
qed

```

```

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'"
  using assms
  by (simp add: alookup_append aupdate_alookup)

```

```

lemma alookup_update_some:
  assumes "alookup xs2 cd2 = Some cd"
    and "alookup xs1 cd1 = Some cd"
  shows "alookup xs2 cd2  $\gg$  ( $\lambda$ cd. aupdate xs1 cd cd1) = Some cd1"

```

```

proof -
  from assms
  have "aupdate xs1 cd cd1 = Some cd1"
  proof (induction rule:aupdate.induct)
    case (1 v uu)
    then show ?case by auto
  next
    case (2 i "is" v xs)
    then show ?case
    apply (case_tac "vtype_class.to_nat i", auto)
    apply (case_tac "xs $ a", auto)
    by (metis list_update_same_conv nth_safe_def option.distinct(1) option.inject)
  next
    case (3 v va uw vb)
    then show ?case by auto
  qed
  then show ?thesis using assms by simp
qed

```

```

lemma alookup_to_nat_same:
  assumes "map to_nat xs = map to_nat ys"
  shows "alookup xs cd = alookup ys cd"
  using assms
  proof (induction xs arbitrary:ys cd)
    case Nil
    then show ?case by simp
  next
    case (Cons a xs)
    then obtain y ys' where "ys = y # ys'" by auto
    show ?case
    proof (cases cd)
      case (Value x1)
      then show ?thesis
      by (simp add: <ys = y # ys'>)
    next
      case (Array xs')
      then show ?thesis
      proof (cases "to_nat a")
        case None
        then have "to_nat y = None" using Cons(2) <ys = y # ys'> by simp
        then show ?thesis using <ys = y # ys'> None Array by simp
      next
        case (Some a')
        then have "to_nat y = Some a'"
        using Cons(2) <ys = y # ys'> by simp
        show ?thesis
      qed
    qed
  qed

```

```

proof (cases "xs' $ a'")
  case None
  then show ?thesis using Some Array <ys = y # ys'> <to_nat y = Some a'>
    by (auto simp add:nth_safe_def split:if_split_asm)
next
  case s: (Some a'')
  then have "alookup xs a'' = alookup ys' a''" using Cons(1) Cons(2)
    by (simp add: <ys = y # ys'>)
  then show ?thesis using Array Some s <to_nat y = Some a'> <ys = y # ys'>
    by (metis bind.bind_lunit alookup.simps(2))
qed
qed
qed
qed

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"
  using assms
  by (metis bind.bind_lunit alookup_append alookup_to_nat_same aupdate_alookup)

lemma aupdate_alookup_nprefix1:
  assumes "xs = ys @ zs"
    and "aupdate xs v cd = Some cd'"
  shows "alookup ys cd' = alookup ys cd  $\gg$  aupdate zs v"
  using assms
proof (induction "xs" arbitrary:cd cd' ys)
  case Nil
  then show ?case by auto
next
  case (Cons i "is'")
  show ?case
  proof (cases ys)
    case Nil
    then show ?thesis
      using Cons.prems(1,2) by auto
  next
    case c: (Cons a list)
    moreover from Cons obtain xs i' i'' cd''
      where 1: "cd=adata.Array xs"
        and 2: "to_nat i = Some i'"
        and 3: "xs $ i' = Some i''"
        and 4: "aupdate is' v i'' = Some cd''"
        and 5: "cd' = adata.Array (list_update xs i' cd'')"
      using aupdate_obtain[of "i # is'" v cd cd']
      by blast
    moreover have "is' = list @ zs"
      using Cons.prems(1) c by auto
    ultimately have "alookup list cd'' = alookup list i''  $\gg$  aupdate zs v" using Cons.IH[of "list", of
      i'' cd''] by blast
    moreover have "a = i"
      using Cons.prems(1) c by auto
    ultimately show ?thesis using c 1 2 3 5 by (auto simp add: nth_safe_def)
  qed
qed

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  $\gg$  aupdate zs v"
  using assms

```

3 Stores and State

```

by (metis alookup_to_nat_same aupdate_alookup_nprefix1)

lemma updateCalldata_clookup_nprefix:
  assumes "aupdate (x#xs) v cd = Some cd'"
    and "to_nat x  $\neq$  to_nat y"
  shows "alookup (y#zs) cd' = alookup (y#zs) cd"
  using assms
proof (cases rule: aupdate_obtain)
  case nil
  then show ?thesis by auto
next
  case (cons i is' xs i' i'' cd'')
  with assms show ?thesis by (cases "vtype_class.to_nat y", auto simp add:nth_safe_def)
qed

lemma aupdate_alookup_nprefix3:
  assumes " $\nexists$ xs'. map to_nat ys = map to_nat xs @ xs'"
    and " $\nexists$ ys'. map to_nat xs = map to_nat ys @ ys'"
    and "aupdate xs v cd = Some cd'"
  shows "alookup ys cd' = alookup ys cd"
  using assms
proof (induction "xs" arbitrary:ys cd cd')
  case Nil
  then show ?case
    by (simp add: prefix_def)
next
  case (Cons i "is'")
  then obtain xs i' i'' cd''
  where 1: "cd=adata.Array xs"
    and 2: "to_nat i = Some i'"
    and 3: "xs $ i' = Some i''"
    and 4: "aupdate is' v i'' = Some cd''"
    and 5: "cd' = adata.Array (list_update xs i' cd'')"
  using aupdate_obtain[of "i # is'" v cd cd']
  by blast

  from Cons have "ys  $\neq$  []" by blast
  then obtain y ys' where "ys = y # ys'" by (meson list.exhaust)
  with Cons(2,3) have " $\neg$  prefix is' ys'  $\wedge$   $\neg$  prefix ys' is'  $\wedge$  to_nat i = to_nat y  $\vee$  to_nat i  $\neq$  to_nat y"
  unfolding prefix_def by auto
  then consider " $\neg$  prefix is' ys'  $\wedge$   $\neg$  prefix ys' is'  $\wedge$  to_nat i = to_nat y" | "to_nat i  $\neq$  to_nat y"
  by blast
  then show ?case
  proof cases
    case *: 1
    then have "  $\nexists$ xs'. map vtype_class.to_nat ys' = map vtype_class.to_nat is' @ xs' " using Cons(2)
    <ys = y # ys'>
    by force
    moreover have "  $\nexists$ xs'. map vtype_class.to_nat is' = map vtype_class.to_nat ys' @ xs' " using
    Cons(3) <ys = y # ys'>
    using "*" by fastforce
    ultimately have "alookup ys' cd'' = alookup ys' i''" using Cons.IH[OF _ _ 4] by blast
    then show ?thesis using <ys = y # ys'> 1 2 3 5 * by (auto simp add: nth_safe_def)
  next
    case 2
    then show ?thesis using updateCalldata_clookup_nprefix[OF Cons(4), of y] <ys = y # ys'> by blast
  qed
qed

lemma alookup_aupdate_some:
  assumes " $\exists$ x. alookup xs cd = Some x"
  shows " $\exists$ x. aupdate xs v cd = Some x"
  using assms
proof (induction xs arbitrary:cd)

```

```

case Nil
then show ?case by simp
next
case (Cons i is')
then obtain cd' i' xs x
  where "cd = adata.Array xs"
  and "vtype_class.to_nat i = Some i'"
  and "xs $ i' = Some cd'"
  and *: "alookup is' cd' = Some x"
  using alookup_obtains_some Cons(2) by blast
moreover from * obtain x'' where "aupdate is' v cd' = Some x''" using Cons(1) by blast
ultimately show ?case by simp
qed

```

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  $\gg$  (\cd. aupdate xs1 cd cd1) = Some cd1"
proof -
  from assms obtain cd1' cd2'
  where *: "aread_safe s1 m l1' = Some cd1'" and "alookup xs1 cd1 = Some cd1'"
  and **: "aread_safe s2 m l2' = Some cd2'" and "alookup xs2 cd2 = Some cd2'"
  using
    read_alookup_obtains[OF assms(4,1)]
    read_alookup_obtains[OF assms(5,2)]
  by blast
  moreover from assms(3) have "cd1' = cd2'" using a_data.read_safe_some_same * ** by blast
  ultimately show ?thesis using alookup_update_some by blast
qed

```

```

lemma split_memory:
  assumes "aread_safe s1 m l = Some cd"
  and "mlookup m xs l = Some l'"
  shows "aread_safe s1 m l'  $\gg$  (\cd'. aupdate xs cd' cd) = Some cd"
  using assms read_alookup_obtains separate_memory by fastforce

```

```

lemma mlookup_read_update:
  assumes "mlookup m0 is l1 = Some l1'"
  and "aread_safe s m0 l1 = Some cd1"
  shows "\exists cd'.
    aupdate is cd cd1 = Some cd'  $\wedge$ 
    (is  $\neq$  []  $\longrightarrow$ 
      (\ls. m0 $ l1 = Some (mdata.Array ls)
         $\wedge$  (\exists as. cd' = adata.Array as  $\wedge$  length as = length ls)))"
  using assms
proof (induction "is" arbitrary: s l1 cd1)
  case Nil
  then show ?case by simp
next
  case (Cons i is')
  then obtain ls i' l'' cd'
  where *: "m0 $ l1 = Some (mdata.Array ls)"
  and **: "to_nat i = Some i'"
  and "ls $ i' = Some l''"
  and "mlookup m0 is' l'' = Some l1'"
  and cd'_def: "aread_safe (finsert l1 s) m0 l'' = Some cd'"
  using a_data.mlookup_read_safe_obtain by metis

  then obtain cd''

```

```

  where "aupdate is' cd cd' = Some cd'" using Cons(1)[of l'' "(finsert l1 s)" cd']
using <mlookup m0 is' l'' = Some l1''> cd'_def by fastforce
moreover from * obtain as
  where "cd1 = Array as"
    and "as $ i' = Some cd'"
    and "length as = length ls"
  using a_data.read_safe_array Cons * cd'_def <ls $ i' = Some l''> by blast
ultimately show ?case using 'to_nat i = Some i'' 'ls $ i' = Some l''' * by simp
qed

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 " $\forall l \in l1 \mid l \notin l1'. m1 \$ l = m0 \$ l$ "
    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"
using assms
proof (induction is1 arbitrary:l1 cd0 cd1 L1 s)
  case Nil
  then show ?case
    by (simp add:case_memory_def split:if_split_asm)
next
  case (Cons i is1')
  from Cons(2)
  obtain ls
    where ls_def: "m0 $ l1 = Some (mdata.Array ls)"
    by (cases is1', auto simp add:case_memory_def split:option.split_asm mdata.split_asm)
  then have m1_ls: "m1 $ l1 = Some (mdata.Array ls)"
    by (metis Cons.prem1(4) assms(3) data.range_safe_subst data.noloops length_list_update
arange_safe_def
  nth_list_update_neq nth_safe_length nth_safe_some)
  moreover from Cons(5) have "l1  $\notin$  s" by auto
  ultimately have
    *: "Some cd1 = those (map (aread_safe (finsert l1 s) m1) ls)  $\ggg$  Some  $\circ$  Array"
    using a_data.read_safe_cases[OF Cons(10)] by fastforce
  moreover obtain as
    where as_def: "aupdate (i # is1') (adata.Value v) cd0 = Some (adata.Array as)"
    and "length as = length ls"
    using mlookup_read_update[OF Cons(2,8)] ls_def by fastforce
  moreover have " $\forall i < \text{length as}. \text{Some } (as!i) = (\text{map } (\text{aread\_safe } (\text{finsert } l1 \text{ s}) m1) ls) ! i$ "
  proof (rule, rule)
    fix i'
    assume "i' < length as"
    then have "i' < length ls" using <length as = length ls> by simp
    then have "(ls ! i')  $\in$  set ls"
      by simp
    show "Some (as ! i') = map (aread_safe (finsert l1 s) m1) ls ! i'"
    proof (cases "to_nat i = Some i'")
      case True
      from Cons(5) <l1  $\notin$  s> ls_def have
        "fold
          ( $\lambda x y. y \ggg (\lambda y'. \text{arange\_safe } (\text{finsert } l1 \text{ s}) m0 x \ggg (\lambda l. \text{Some } (l \mid \cup y'))$ ))
          ls
          (Some (l1))
        = Some L1"
        by (simp add: case_memory_def split:if_split_asm)
      then obtain LL
        where LL_def: "arange_safe (finsert l1 s) m0 (ls ! i') = Some LL"
        and "LL  $\subseteq$  L1"

```

```

using ls_def True <i' <length ls> <l1 |∉| s>
fold_some_subst[of "arange_safe (finsert l1 s) m0" ls "Some {|l1|}" L1 "ls ! i'"] by auto
moreover from Cons(2) have range_the: "locations m0 (i # is1') l1 = Some (the (locations m0 (i
# is1') l1))"
by (metis mlookup_locations_some option.sel)
then obtain LLL
where LLL_def: "locations m0 is1' (ls ! i') = Some LLL"
and LLL_L3: "(the (locations m0 (i # is1') l1)) = finsert l1 LLL"
using <l1 |∉| s> ls_def locations_obtain True
by (metis <i' <length ls> mdata.inject(2) nth_safe_some option.sel)
from Cons(2) have ll_def: "mlookup m0 is1' (ls!i') = Some l1'"
using <l1 |∉| s> ls_def True
apply (cases " marray_lookup m0 (i # is1') l1",auto)
by (metis Cons.prem1 <i' <length ls> mdata.inject(2) mlookup_obtain_empty2
nth_safe_some option.inject)

from Cons(8) have
"those (map (aread_safe (finsert l1 s) m0) ls) ≧= Some o adata.Array = Some cd0"
using <l1 |∉| s> ls_def using True
by (auto simp add:case_memory_def split:if_split_asm)
then obtain cd1'
where cd1'_def: "aread_safe (finsert l1 s) m0 (ls!i') = Some cd1'"
using <l1 |∉| s> ls_def True <(ls ! i') ∈ set ls> LL_def a_data.range_safe_read_safe
by blast

from Cons(10) have
"those (map (aread_safe (finsert l1 s) m1) ls) ≧= Some o adata.Array = Some cd1"
using <l1 |∉| s> m1_ls True
by (auto simp add:case_memory_def split:if_split_asm)
then obtain cd2'
where cd2'_def: "aread_safe (finsert l1 s) m1 (ls!i') = Some cd2'"
using those_map_none[of "aread_safe (finsert l1 s) m1" ls] <(ls ! i') ∈ set ls>
by fastforce
thm Cons(1)
moreover have "aupdate is1' (Value v) cd1' = Some cd2'"
proof (rule Cons(1)[OF
ll_def
Cons(3,4)
LL_def
-
-
cd1'_def
-
cd2'_def
])
have "l1 |∉| L1'"
using Cons.prem1(1,4,5) a_data.noloops by blast
then show "arange_safe (finsert l1 s) m0 l1' = Some L1'" using Cons(6)
by (smt (verit, best) finsertE fminusD1 fminusD2 a_data.range_safe_nin_same)
next
from Cons(7) show "∀l|∈|LL |-| L1'. m1 $ l = m0 $ l" using <LL |⊆| L1> by blast
next
from Cons(9) show "adisjoint m0 LL" using <LL |⊆| L1> by auto
qed
then have "(as ! i') = cd2'"
using as_def cd1'_def ls_def True Cons(8)
apply (auto simp add:case_memory_def split:if_split_asm)
apply (cases cd0,auto)
apply (case_tac "x2 $ i'",auto)
apply (case_tac " aupdate is1' (adata.Value v) a",auto)
apply (case_tac "those (map (aread_safe (finsert l1 s) m0) ls)",auto)
apply (case_tac "m0 $ (ls ! i')",auto)
apply (case_tac "ab",auto)
using <i' <length as> aupdate_obtain apply fastforce

```

```

    by (metis <i' < length as> cd1'_def length_list_update map_equality_iff
        nth_list_update_eq nth_safe_some option.inject those_some_map)
  moreover have
    "map (aread_safe (finsert l1 s) m1) ls ! i'
    = aread_safe (finsert l1 s) m1 (ls ! i')"
    by (simp add: <i' < length ls>)
  ultimately show ?thesis by simp
next
case False
then obtain i''
  where "to_nat i = Some i''"
    and "i'' ≠ i'"
    and "i'' < length ls"
  using Cons(2) ls_def
  apply (case_tac "to_nat i", auto)
  apply (cases is1', auto simp add: case_memory_def)
  using mlookup_obtain_nempty2 by fastforce
moreover from Cons(8) ls_def have
  *: "those (map (aread_safe (finsert l1 s) m0) ls) ≫= Some o Array = Some cd0"
  using <l1 |∈| s> by (auto simp add: case_memory_def)
moreover from * obtain aa
  where aa_def: "those (map (aread_safe (finsert l1 s) m0) ls) = Some aa"
  by (cases "those (map (aread_safe (finsert l1 s) m0) ls)", auto)
moreover from aa_def have "length aa = length ls" by (metis length_map those_some_map)
then have "aread_safe (finsert l1 s) m0 (ls ! i') = Some (aa!i')"
  using * those_map_nth[of "aread_safe (finsert l1 s) m0" ls aa i']
  aa_def 'i' < length ls'
  by (auto simp add: nth_safe_def split:if_split_asm option.split_asm)
ultimately have "aread_safe (finsert l1 s) m0 (ls ! i') = Some (as ! i')"
  using aupdate_nth_same[of i is1' "Value v" aa as i'' i'] as_def by force
moreover from Cons(5) have
  *: "fold
    (λx y. y ≫= (λy'. (arange_safe (finsert l1 s) m0 x) ≫= (λl. Some (l |∪| y'))))
    ls
    (Some {|l1|})
  = Some L1" using ls_def
  by (auto simp add: case_memory_def split:if_split_asm)
then obtain L
  where L_def: "arange_safe (finsert l1 s) m0 (ls ! i') = Some L"
    and "L |⊆| L1"
  using fold_some_subs[of "arange_safe (finsert l1 s) m0"]
  by (meson <ls ! i' ∈ set ls>)
moreover have "∀ l |∈| L. m1 $ l = m0 $ l"
proof -
  from L_def have "arange m0 (ls ! i') = Some L"
    unfolding arange_def arange_safe_def data.range_def
    using data.range_safe_subset_same by blast
  moreover have "ls ! i'' ∈ set ls" by (simp add: <i'' < length ls>)
  with * have "∃ L'. arange_safe (finsert l1 s) m0 (ls ! i'') = Some L'"
    using fold_some_subs[of "arange_safe (finsert l1 s) m0"]
    by (meson)
  then obtain L' where L'_def: "arange_safe s m0 (ls ! i'') = Some L'"
    using a_data.range_safe_subset_same by blast
  then have "arange m0 (ls ! i'') = Some L'"
    unfolding arange_def arange_safe_def data.range_def
    using data.range_safe_subset_same by blast
  moreover from <ls ! i' ∈ set ls> have "ls $ i' = Some (ls ! i')"
    unfolding nth_safe_def using <i' < length ls> by simp
  moreover have "ls $ i'' = Some (ls ! i'')"
    unfolding nth_safe_def by (simp add: <i'' < length ls>)
  moreover have "l1 |∈| L1"
    using Cons.premis(4) a_data.range_safe_subs by auto
  ultimately have "(L |∩| L' = {|})"
    using Cons(9) unfolding a_data.disjoint_def

```

```

    using ls_def 'i' ≠ i' by blast
  moreover have "mlookup m0 is1' (ls ! i') = Some l1'"
    using Cons(2) ls_def <to_nat i = Some i''>
    using <ls $ i'' = Some (ls ! i'')> mlookup_obtain_nempty2 by fastforce
  then have "L1' |⊆| L'" using a_data.mlookup_range_safe_subs[OF _ Cons(6) L'_def] L'_def by
simp
    ultimately have "L |∩| L1' = {||}" by blast
    then show ?thesis using 'L |⊆| L1' Cons(7) by auto
  qed
  ultimately have "aread_safe (finsert l1 s) m1 (ls ! i') = Some (as ! i')"
    using a_data.read_safe_range_safe[of "finsert l1 s" m0 "ls!i'" _ L m1] by blast
  then show ?thesis using <i' < length ls> by auto
  qed
  qed
  ultimately show ?case by (simp add:map_some_those_some)
  qed

```

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"
  using assms
proof (induction is1 arbitrary:l1 cd cd1 L1 s)
  case Nil
  moreover have "mlookup m1 is2 l2 = Some l2'"
  proof -
    from Nil have "l2' |∈| L2" using a_data.range_safe_mlookup by blast
    then have "m1 $ l2' = m0 $ l2'" using Nil(8) by simp
    moreover from Nil(2) obtain L where "locations m0 is2 l2 = Some L"
      using mlookup_locations_some by blast
    then have "L |⊆| L2" using a_data.range_safe_locations[OF Nil(6)] by simp
    then have "∀ l |∈| L. m1 $ l = m0 $ l" using Nil(8) by blast
    ultimately show ?thesis
      using mlookup_same_locations[OF Nil(2) 'locations m0 is2 l2 = Some L'] by auto
  qed
  moreover from Nil have "m1$l1 = m0$l2'" by simp
  then have "m1$l2' = m1$l1" using Nil a_data.range_safe_mlookup by metis
  then have "m1 $ l1' = m1 $ l2'" by (metis bind.bind_lunit calculation(1) mlookup.simps(1))
  moreover have "aread_safe s2 m1 l2 = Some cd2"
    using a_data.read_safe_range_safe[OF Nil(10,6,8)] .
  ultimately show ?case using separate_memory[of m1 "[]" l1 l1' is2 l2 l2' s cd s2 cd2] by auto
next
  case (Cons i is1')
  from Cons(2)
  obtain ls
    where ls_def: "m0 $ l1 = Some (mdata.Array ls)"
      by (cases is1', auto simp add:case_memory_def split:option.split_asm mdata.split_asm)
  then have m1_ls: "m1 $ l1 = Some (mdata.Array ls)"
    by (metis Cons.prem1(4,7) assms(5) a_data.range_safe_subs fminus_iff a_data.noloops
      a_data.range_safe_in_subs)
  moreover from Cons(13) have "l1 |∉| s" by auto
  ultimately have
    *: "Some cd = those (map (aread_safe (finsert l1 s) m1) ls) ≧≧ Some ∘ Array"
    using a_data.read_safe_cases[OF Cons(13)] by fastforce

```

```

moreover obtain as
  where as_def: "alookup is2 cd2
    >>= (λcd. aupdate (i # is1') cd cd1) = Some (adata.Array as)"
    and "length as = length ls"
proof -
  from Cons(3,11) obtain cd'
    where "alookup is2 cd2 = Some cd'"
    using mlookup_read_alookup by blast
  moreover obtain as
    where "aupdate (i # is1') cd' cd1 = Some (adata.Array as)"
    and "length as = length ls"
    using mlookup_read_update[OF Cons(2,10)] ls_def by fastforce
  ultimately show ?thesis using that by simp
qed
moreover have "∀i < length as. Some (as!i) = (map (aread_safe (finsert l1 s) m1) ls) ! i"
proof (rule, rule)
  fix i'
  assume "i' < length as"
  then have "i' < length ls" using <length as = length ls> by simp
  then have "(ls ! i') ∈ set ls"
    by simp
  show "Some (as ! i') = map (aread_safe (finsert l1 s) m1) ls ! i'"
  proof (cases "to_nat i = Some i'")
    case True
    from Cons(5) <l1 |∉| s> ls_def have
      "fold
        (λx y. y >>= (λy'. arange_safe (finsert l1 s) m0 x >>= (λl. Some (l |∪| y'))))
        ls
        (Some ({|l1|}))
      = Some L1"
    by (simp add: case_memory_def split:if_split_asm)
    then obtain LL
      where LL_def: "arange_safe (finsert l1 s) m0 (ls ! i') = Some LL"
      and "LL |⊆| L1"
    using ls_def True <i' < length ls> <l1 |∉| s>
      fold_some_subst[of "arange_safe (finsert l1 s) m0" ls "Some {|l1|}" L1 "ls ! i'"] by auto

  from Cons(2) obtain L3 where L3_def: "locations m0 (i # is1') l1 = Some L3"
  using mlookup_locations_some by blast
  obtain LLL
    where LLL_def: "locations m0 is1' (ls ! i') = Some LLL"
    and LLL_L3: "L3 = finsert l1 LLL"
  using <l1 |∉| s> ls_def locations_obtain[OF L3_def] True
  by (metis <i' < length ls> mdata.inject(2) nth_safe_some option.sel)
  from Cons(2) have ll_def: "mlookup m0 is1' (ls!i') = Some ll'"
  using <l1 |∉| s> ls_def True
  apply (cases "marray_lookup m0 (i # is1') l1", auto)
  by (metis Cons.prem1 <i' < length ls> mdata.inject(2) mlookup_obtain_empty2
    nth_safe_some option.inject)

  from Cons(10) have
    "those (map (aread_safe (finsert l1 s) m0) ls) >>= Some o adata.Array = Some cd1"
  using <l1 |∉| s> ls_def using True
  by (auto simp add: case_memory_def split:if_split_asm)
  then obtain cd1'
    where cd1'_def: "aread_safe (finsert l1 s) m0 (ls!i') = Some cd1'"
  using <l1 |∉| s> ls_def True <(ls ! i') ∈ set ls> LL_def a_data.range_safe_read_safe
  by blast

  from Cons(13) have
    "those (map (aread_safe (finsert l1 s) m1) ls) >>= Some o adata.Array = Some cd"
  using <l1 |∉| s> m1_ls True
  by (auto simp add: case_memory_def split:if_split_asm)
  then obtain cd2'

```

```

where cd2'_def: "aread_safe (finsert l1 s) m1 (ls!i') = Some cd2'"
using those_map_none[of "aread_safe (finsert l1 s) m1" ls] <(ls ! i') ∈ set ls>
by fastforce

moreover have "alookup is2 cd2 ≧≧ (λcd. aupdate is1' cd cd1') = Some cd2'"
proof (rule Cons(1)[OF
  l1_def
  Cons(3,4)
  LL_def
  -
  Cons(7)
  -
  Cons(9)
  cd1'_def
  Cons(11)
  -
  cd2'_def])
  have "l1 |∉| L1'"
    using Cons.prem(1,4,5) a_data.noloops by blast
  then show "arange_safe (finsert l1 s) m0 l1' = Some L1'" using Cons(6)
    by (smt (verit, best) finsertE fminusD1 fminusD2 a_data.range_safe_nin_same)
next
  from Cons(8) show "∀l|∈|LL |-| L1'. m1 $ l = m0 $ l" using <LL |⊆| L1> by blast
next
  from Cons(12) show "adisjoint m0 LL" using <LL |⊆| L1> by auto
qed
then have "(as ! i') = cd2'"
  using as_def cd1'_def ls_def True Cons(10)
  apply (auto simp add:case_memory_def split:if_split_asm)
  apply (cases " alookup is2 cd2",auto)
  apply (cases cd1,auto)
  apply (case_tac "x2 $ i'",auto)
  apply (case_tac " aupdate is1' a aa",auto)
  apply (case_tac "those (map (aread_safe (finsert l1 s) m0) ls)",auto)
  apply (case_tac "m0 $ (ls ! i')",auto)
  apply (case_tac "ac",auto)
  using <i' < length as> aupdate_obtain apply fastforce
  by (metis <i' < length as> cd1'_def length_list_update map_equality_iff
    nth_list_update_eq nth_safe_some option.inject those_some_map)
moreover have
  "map (aread_safe (finsert l1 s) m1) ls ! i'
  = aread_safe (finsert l1 s) m1 (ls ! i')"
  by (metis L3_def True <m0 $ l1 = Some (mdata.Array ls)>
    locations_obtain mdata.inject(2) nth_map nth_safe_length option.inject)
ultimately show ?thesis by simp
next
case False
then obtain i''
  where "to_nat i = Some i''"
    and "i'' ≠ i'"
    and "i'' < length ls"
  using Cons(2) ls_def
  apply (case_tac "to_nat i",auto)
  apply (cases is1',auto simp add:case_memory_def)
  using mlookup_obtain_nempty2 by fastforce
moreover from Cons(10) ls_def have
  *: "those (map (aread_safe (finsert l1 s) m0) ls) ≧≧ Some o Array = Some cd1"
  using <l1 |∉| s> by (auto simp add:case_memory_def)
moreover from * obtain aa
  where aa_def: "those (map (aread_safe (finsert l1 s) m0) ls) = Some aa"
  by (cases "those (map (aread_safe (finsert l1 s) m0) ls)", auto)
moreover from aa_def have "length aa = length ls" by (metis length_map those_some_map)
then have "aread_safe (finsert l1 s) m0 (ls ! i') = Some (aa!i'"
  using * those_map_nth[of "aread_safe (finsert l1 s) m0" ls aa i']

```

```

aa_def 'i' < length ls'
by (auto simp add:nth_safe_def split:if_split_asm option.split_asm)
moreover from Cons(3,11) obtain cd'
  where "alookup is2 cd2 = Some cd'"
  using mlookup_read_alookup by blast
ultimately have "aread_safe (finsert l1 s) m0 (ls ! i') = Some (as ! i')"
  using aupdate_nth_same[of i is1' cd' aa as i'' i'] as_def by force
moreover from Cons(5) have
  *:"fold
    (λx y. y ≫= (λy'. (arange_safe (finsert l1 s) m0 x) ≫= (λl. Some (l |U| y'))))
    ls
    (Some {|l1|})
  = Some L1" using ls_def
by (auto simp add:case_memory_def split:if_split_asm)
then obtain L
  where L_def: "arange_safe (finsert l1 s) m0 (ls ! i') = Some L"
  and "L |⊆| L1"
  using fold_some_subs[of "arange_safe (finsert l1 s) m0"]
  by (meson <ls ! i' ∈ set ls>)
moreover have "∀l|∈|L. m1 $ l = m0 $ l"
proof -
  from L_def have "arange m0 (ls ! i') = Some L"
  unfolding arange_def arange_safe_def data.range_def
  using data.range_safe_subset_same by blast
  moreover have "ls ! i'' ∈ set ls" by (simp add: <i'' < length ls>)
  with * have "∃L'. arange_safe (finsert l1 s) m0 (ls ! i'') = Some L'"
  using fold_some_subs[of "arange_safe (finsert l1 s) m0"]
  by (meson)
  then obtain L' where L'_def: "arange_safe s m0 (ls ! i'') = Some L'"
  using a_data.range_safe_subset_same by blast
  then have "arange m0 (ls ! i'') = Some L'"
  unfolding arange_def arange_safe_def data.range_def
  using data.range_safe_subset_same by blast
  moreover from <ls ! i' ∈ set ls> have "ls $ i' = Some (ls ! i')"
  unfolding nth_safe_def using <i' < length ls> by simp
  moreover have "ls $ i'' = Some (ls ! i'')"
  unfolding nth_safe_def by (simp add: <i'' < length ls>)
  moreover have "l1 |∈| L1"
  using Cons.premis(4) a_data.range_safe_subs by auto
  ultimately have "(L |∩| L' = {|})"
  using Cons(12) unfolding a_data.disjoint_def
  using ls_def 'i'' ≠ i'' by blast
  moreover have "mlookup m0 is1' (ls ! i'') = Some l1'"
  using Cons(2) ls_def <to_nat i = Some i''>
  using <ls $ i'' = Some (ls ! i'')> mlookup_obtain_nempty2 by fastforce
  thm Cons(6)
  then have "L1' |⊆| L'" using a_data.mlookup_range_safe_subs[OF _ Cons(6) L'_def] L'_def by
simp
ultimately have "L |∩| L1' = {|}" by blast
then show ?thesis using 'L |⊆| L1' Cons(8) by auto
qed
ultimately have "aread_safe (finsert l1 s) m1 (ls ! i') = Some (as ! i')"
  using a_data.read_safe_range_safe[of "finsert l1 s" m0 "ls!i'" _ L m1] by blast
then show ?thesis using <i' < length ls> by auto
qed
qed
ultimately show ?case by (simp add:map_some_those_some)
qed

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 "( $\forall l \in l1 \mid \neg l1'.$  m1 $ l = m0 $ l)"
and "( $\forall l \in l2'.$  m1 $ l = m0 $ l)"
and "adisjoint m0 l1"
and "s | $\cap$ | l2' = {}"
and "the (locations m0 is1 l1) | $\cap$ | l2' = {}"
and "l1' | $\notin$ | l2'"
shows " $\exists L.$  arange_safe s m1 l1 = Some L"
using assms
proof (induction is1 arbitrary:l1 l1 s)
case Nil
then have "l1 = l1'" by auto
then have "m1 $ l1 = m0 $ l2'" using Nil by simp

have "l1 | $\notin$ | s" using Nil(3) by (simp add:case_memory_def split: if_split_asm)
have "l2' | $\notin$ | s2" using Nil(5) by (simp add:case_memory_def split: if_split_asm)

show ?case
proof (cases "m1 $ l1")
case None
then show ?thesis
by (metis <m1 $ l1 = m0 $ l2'> assms(5) mlookup.simps(1) not_None_eq
a_data.mlookup_range_safe_some)
next
case (Some a)
then show ?thesis
proof (cases a)
case (Value v)
then show ?thesis using <l1 | $\notin$ | s> Some by (auto simp add:case_memory_def)
next
case (Array xs)
moreover have " $\forall x \in \text{set } xs.$  arange_safe (finsert l1 s) m1 x  $\neq$  None"
proof
fix x
assume "x  $\in$  set xs"
moreover have "fold
( $\lambda x y. y \gg (\lambda y'. (\text{arange\_safe } (\text{finsert } l2' s2) m0 x) \gg (\lambda l. \text{Some } (l \cup y'))))$ 
xs (Some {l2'}) = Some L2'"
using Nil(5) <l2' | $\notin$ | s2> <m1 $ l1 = m0 $ l2'> Some Array
by (auto simp add:case_memory_def split:option.split_asm)
ultimately obtain X
where "arange_safe (finsert l2' s2) m0 x = Some X"
and "X | $\subseteq$ | l2'"
by (metis Array Some <m1 $ l1 = m0 $ l2'> assms(5) a_data.range_safe_obtains_subset)
then have "arange_safe (finsert l2' s2) m1 x = Some X"
using Nil(7) a_data.range_safe_same[of "(finsert l2' s2)" m0 x X m1]
by blast
moreover have " $\forall l \in l1$  (finsert l1 s) - (finsert l2' s2). l | $\notin$ | X"
proof -
have "l1 | $\notin$ | X" using Nil.prem(11) <X | $\subseteq$ | l2'> <l1 = l1'> by auto
then show ?thesis using Nil(9) using <X | $\subseteq$ | l2'> by blast
qed
ultimately have "arange_safe (finsert l1 s) m1 x = Some X" using a_data.range_safe_nin_same[of
"(finsert l2' s2)" m1 x X] by blast
then show "arange_safe (finsert l1 s) m1 x  $\neq$  None" by simp
qed
then have "fold
( $\lambda x y. y \gg (\lambda y'. (\text{arange\_safe } (\text{finsert } l1 s) m1 x) \gg (\lambda l. \text{Some } (l \cup y'))))$ 
xs (Some {l1})  $\neq$  None"
using fold_f_set_some[of _ "arange_safe (finsert l1 s) m1"] by blast
ultimately show ?thesis using Some <l1 | $\notin$ | s> by (auto simp add:case_memory_def)
qed
qed
next

```

```

case (Cons i is1)
have "l1 |∉| s" using Cons(4) by (simp add:case_memory_def split: if_split_asm)
have "l2' |∉| s2" using Cons(6) by (simp add:case_memory_def split: if_split_asm)

then show ?case
proof (cases "m1 $ l1")
  case None
  then show ?thesis
  by (metis Cons.prem1(1,3,4,6) fminus_iff mlookup_obtain_nempty2 option.discI
a_data.range_safe_subs
  a_data.noloops)
next
case (Some a)
then show ?thesis
proof (cases a)
  case (Value x1)
  moreover obtain xs where "m0$l1 = Some (mdata.Array xs)"
  using mlookup_obtain_nempty2[OF Cons(2)] by auto
  moreover have "l1 |∉| L1'"
  using Cons.prem1(1,3,4) a_data.noloops by blast
  ultimately show ?thesis using Cons(7)
  by (metis Cons.prem1(3) Some fminusI mdata.distinct(1) option.inject a_data.range_safe_subs)
next
case (Array xs)
moreover have "∀x ∈ set xs. arange_safe (finsert l1 s) m1 x ≠ None"
proof
  fix x assume "x ∈ set xs"
  then obtain i' where x_def: "xs $ i' = Some x"
  by (meson set_nth_some)
  have m1_ls: "m0 $ l1 = Some (mdata.Array xs)"
  by (metis Array Cons.prem1(1,3,4,6) Some data.range_safe_subs data.noloops fminus_iff
arange_safe_def)
  then obtain j' l'' where "to_nat i = Some j'" and l''_def: "xs $ j' = Some l''" and *:
"mlookup m0 is1 l'' = Some l1'"
  using mlookup_obtain_nempty2[OF Cons(2)]
  by (metis mdata.inject(2) option.inject)

  obtain L' where L'_def: "arange_safe (finsert l1 s) m0 l'' = Some L'" and "L' |⊆| L1"
  by (meson Cons.prem1(3) <xs $ j' = Some l''> m1_ls nth_in_set a_data.range_safe_obtains_subset)

  from Cons(2) obtain LL where LL_def: "locations m0 (i # is1) l1 = Some LL"
  using mlookup_locations_some by blast
  show "arange_safe (finsert l1 s) m1 x ≠ None"
  proof (cases "i' = j'")
    case True
    then have "x = l''"
    using <xs $ i' = Some x> <xs $ j' = Some l''> by auto
    moreover have "l1 |∉| L1'"
    using Cons.prem1(1,3,4) a_data.noloops by blast
    with Cons(5) have "arange_safe (finsert l1 s) m0 l1' = Some L1'" using
a_data.range_safe_nin_same[of s m0 l1' L1' "finsert l1 s"] by blast
    moreover from Cons(7) have "∀l|∈|L' |∉| L1'. m1 $ l = m0 $ l" using <L' |⊆| L1> by blast
    moreover from Cons(9) have "adisjoint m0 L'" using <L' |⊆| L1> by auto
    moreover from Cons(2,10,11) have "finsert l1 s |∩| L2' = {}"
    proof -
      from LL_def have "l1 |∈| LL" using locations_l_in_L by blast
      then show ?thesis using Cons(10,11) LL_def by auto
    qed
    moreover have "the (locations m0 is1 l'') |∩| L2' = {}"
    proof -
      from LL_def obtain LLL where "locations m0 is1 l'' = Some LLL"
      using "*" mlookup_locations_some by blast
      moreover have "mlookup m0 [i] l1 = Some l''" using * <to_nat i = Some j''> l''_def m1_ls
      by (auto simp add:case_memory_def)

```

```

ultimately have "LLL |⊆| LL" using mlookup_locations_subs[of m0 "[i]"] LL_def by auto
then show ?thesis using Cons(11) LL_def <locations m0 is1 l'' = Some LLL> by auto
qed
ultimately show ?thesis using Cons(1)[OF * Cons(3) L'_def _ Cons(6) _ Cons(8) _ _ _
Cons(12)] by blast
next
case False
moreover have "l1 |∈| L1"
  using Cons.prem3 a_data.range_safe_subs by auto
moreover obtain L where L_def: "arange_safe (finsert l1 s) m0 x = Some L" and "L |⊆| L1"
using Cons(4) Some m1_ls <l1 |∉| s>
  by (metis <x ∈ set xs> data.range_safe_obtains_subset arange_safe_def)
then have "arange m0 x = Some L" unfolding a_data.range_def
  using a_data.range_safe_subset_same by blast
moreover have "L1' |⊆| L'" using Cons(5) L'_def a_data.mlookup_range_safe_subs[OF *, of s _
"(finsert l1 s)"] by blast
moreover from L'_def have "arange m0 l'' = Some L'" unfolding a_data.range_def
  using a_data.range_safe_subset_same by blast
ultimately have "L |∩| L1' = {}" using Cons(9) unfolding a_data.disjoint_def using m1_ls
x_def l''_def by blast
then show ?thesis using <L |⊆| L1> Cons(7) a_data.range_safe_same[of "finsert l1 s" m0 x L
m1]
  L_def by blast
qed
qed
then have "fold
  (λx y. y ≧ (λy'. (arange_safe (finsert l1 s) m1 x) ≧ (λl. Some (l |∪| y'))))
  xs (Some {l1}) ≠ None"
using fold_f_set_some[of _ "arange_safe (finsert l1 s) m1"] by blast
ultimately show ?thesis using Some <l1 |∉| s> by (auto simp add:case_memory_def)
qed
qed
qed
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"
using assms unfolding a_data.range_def
using range_safe_update_some by blast

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'"
using assms
proof (induction is1 arbitrary:l1 L L1 s)
case Nil
then have "l1 = l1'" by auto

```

```

then have "m1 $ l1 = m0 $ l2'" using Nil by simp

have "l1 | $\notin$ | s" using Nil(3) by (simp add:case_memory_def split: if_split_asm)
have "l2' | $\notin$ | s2" using Nil(5) by (simp add:case_memory_def split: if_split_asm)

show ?thesis
proof (cases "m1 $ l1")
  case None
  then show ?thesis
  by (metis Nil.prem1(9) data.range_safe_obtains arange_safe_def option.distinct(1))
next
case (Some a)
then show ?thesis
proof (cases a)
  case (Value v)
  then have "L = {|l1|}" using Nil(9) <l1 | $\notin$ | s> Some by (auto simp add:case_memory_def)
  moreover have "l1 | $\in$ | L1" using Nil(3) using a_data.range_safe_subs[of s m0 l1 L1] by simp
  ultimately show ?thesis by simp
next
case (Array xs)
show ?thesis
proof
  fix x assume "x | $\in$ | L"
  moreover have "fold
    (\x y. y  $\gg$  (\y'. (arange_safe (finsert l1 s) m1 x)  $\gg$  (\lambda l. Some (l | $\cup$ | y'))))
    xs (Some {|l1|}) = Some L"
  using Nil(9) using <l1 | $\notin$ | s> Some Array by (auto simp add:case_memory_def)
  ultimately consider
    (1) "x = l1"
  | (2) n L'' where "n < length xs  $\wedge$  arange_safe (finsert l1 s) m1 (xs ! n) = Some L''  $\wedge$  x | $\in$ |
    L''"
  using fold_union_in[of "arange_safe (finsert l1 s) m1"] by blast
  then show "x | $\in$ | L1 | $\cup$ | L2'"
  proof cases
    case 1
    moreover have "l1 | $\in$ | L1" using Nil(3) using a_data.range_safe_subs[of s m0 l1 L1] by simp
    ultimately show ?thesis by simp
  next
    case 2
    moreover have "L'' | $\subseteq$ | L2'"
    proof -
      have "fold
        (\x y. y  $\gg$  (\y'. (arange_safe (finsert l2' s2) m0 x)  $\gg$  (\lambda l. Some (l | $\cup$ | y'))))
        xs (Some {|l2'|}) = Some L2'"
      using Nil(5) <l2' | $\notin$ | s2> <m1 $ l1 = m0 $ l2'> Some Array
      by (auto simp add:case_memory_def split:option.split_asm)
      then obtain X
      where "arange_safe (finsert l2' s2) m0 (xs ! n) = Some X"
      and "X | $\subseteq$ | L2'" using fold_some_subs
      by (metis "2" nth_mem)
      then have "arange_safe (finsert l2' s2) m1 (xs ! n) = Some X"
      using Nil(7) a_data.range_safe_same[of "(finsert l2' s2)" m0 "(xs ! n)" X m1]
      by blast
      then have "arange_safe (finsert l1 s) m1 (xs ! n) = Some X"
      using "2" a_data.range_range by blast
      then show ?thesis
      using <X | $\subseteq$ | L2'> a_data.range_range "2" by auto
    qed
  qed
  ultimately show ?thesis by blast
qed
qed
qed
qed
qed
next

```

```

case (Cons i is1)
have "l1 |∉| s" using Cons(4) by (simp add:case_memory_def split: if_split_asm)
have "l2' |∉| s2" using Cons(6) by (simp add:case_memory_def split: if_split_asm)

then show ?case
proof (cases "m1 $ l1")
  case None
  then show ?thesis
  by (metis Cons.prem(9) option.distinct(1) a_data.read_safe_cases a_data.range_safe_read_safe)
next
case (Some a)
then show ?thesis
proof (cases a)
  case (Value x1)
  moreover obtain xs where "m0$l1 = Some (mdata.Array xs)"
  using mlookup_obtain_nempty2[OF Cons(2)] by auto
  moreover have "l1 |∈| L" using Cons(4)
  using Cons.prem(9) a_data.range_safe_subs by auto
  moreover have "l1 |∉| L1'"
  using Cons.prem(1,3,4) a_data.noloops by blast
  ultimately show ?thesis using Cons(7)
  by (metis Cons.prem(3) Some fminusI mdata.distinct(1) option.inject a_data.range_safe_subs)
next
case (Array xs)
then have "fold
  (λx y. y ≧ (λy'. (arange_safe (finsert l1 s) m1 x) ≧ (λl. Some (l |∪| y'))))
  xs (Some {|l1|}) = Some L"
  using Cons(10) using <l1 |∉| s> Some by (auto simp add:case_memory_def)
  moreover have "∀x ∈ set xs. ∀L. arange_safe (finsert l1 s) m1 x = Some L → L |⊆| L1 |∪|
L2'"
proof
  fix x assume "x ∈ set xs"
  then obtain i' where "xs $ i' = Some x"
  by (meson set_nth_some)
  have m1_ls: "m0 $ l1 = Some (mdata.Array xs)"
  by (metis Array Cons.prem(1,3,4,6) Some data.range_safe_subs data.noloops fminus_iff
arange_safe_def)
  then obtain j' l'' where "to_nat i = Some j'" and "xs $ j' = Some l''" and *: "mlookup m0
is1 l'' = Some l1'"
  using mlookup_obtain_nempty2[OF Cons(2)]
  by (metis mdata.inject(2) option.inject)
  show "∀L. arange_safe (finsert l1 s) m1 x = Some L → L |⊆| L1 |∪| L2'"
  proof (cases "i' = j'")
    case True
    then have "x = l1'"
    using <xs $ i' = Some x> <xs $ j' = Some l1''> by auto
    show ?thesis
    proof (rule allI, rule impI)
      fix L
      assume L_def: "arange_safe (finsert l1 s) m1 x = Some L"
      obtain LL where LL_def: "arange_safe (finsert l1 s) m0 l1'' = Some LL" and "LL |⊆| L1"
using Cons(4) *
      by (metis True <x ∈ set xs> <xs $ i' = Some x> <xs $ j' = Some l1''>
data.range_safe_obtains_subset arange_safe_def m1_ls option.inject)
      moreover have "L |⊆| LL |∪| L2'"
      proof (rule Cons(1)[OF * Cons(3) LL_def _ Cons(6) _ Cons(8)])
        have "l1 |∉| L1'"
        by (metis Cons.prem(1,3,4) data.noloops arange_safe_def)
        with Cons(5) show "arange_safe (finsert l1 s) m0 l1'' = Some L1'" using <l1 |∉| L1''>
        by (smt (verit, best) finsertE fminusD1 fminusD2 a_data.range_safe_nin_same)
      next
      from Cons(7) show "∀l|∈|LL |-| L1'. m1 $ l = m0 $ l" using <LL |⊆| L1> by auto
      next
      from Cons(9) show "adisjoint m0 LL"

```

```

    using calculation(2) by auto
  next
    from L_def show "arange_safe (finsert l1 s) m1 l'' = Some L" using <x = l''> by simp
  qed
  ultimately show "L |⊆| L1 |∪| L2'" by auto
qed
next
case False
show ?thesis
proof (rule allI, rule impI)
  fix L
  assume L_def: "arange_safe (finsert l1 s) m1 x = Some L"
  moreover from Cons(4) have "fold
    (λx y. y ≧ (λy'. (arange_safe (finsert l1 s) m0 x) ≧ (λl. Some (l |∪| y'))))
    xs (Some {l1}) = Some L1" using m1_ls <l1 |∉| s> by (simp add:case_memory_def)
  then obtain L' where L'_def: "arange_safe (finsert l1 s) m0 x = Some L'" and "L' |⊆| L1"
    by (metis Cons.prem(3) <x ∈ set xs> data.range_safe_obtains_subset arange_safe_def
m1_ls)
  moreover have "L = L'"
  proof -
    have "l1 ∈ L1"
      by (metis Cons.prem(3) data.range_safe_subs arange_safe_def)
    moreover have "arange m0 x = Some L'" using L'_def
      by (metis bot.extremum a_data.range_def a_data.range_safe_subset_same)
    moreover obtain LL where LL_def: "arange_safe (finsert l1 s) m0 l'' = Some LL"
      by (meson Cons.prem(3) <xs $ j' = Some l''> m1_ls nth_in_set
a_data.range_safe_obtains_subset)
    then have "L1' |⊆| LL" using Cons(5) a_data.mlookup_range_safe_subs[OF *, of s _
"(finsert l1 s)"] by blast
    moreover from LL_def have "arange m0 l'' = Some LL"
      by (metis all_not_fin_conv data.range_safe_nin_same empty_fminus arange_safe_def
a_data.range_def)
    ultimately have "L' |∩| L1' = {}" using Cons(9) unfolding a_data.disjoint_def
      using m1_ls <L' |⊆| L1> False <xs $ i' = Some x> <xs $ j' = Some l''> by blast
    then have "∀ l ∈ L'. m1 $ l = m0 $ l" using Cons(7) 'L' |⊆| L1' by auto
    then show ?thesis using a_data.range_safe_same[OF L'_def] using L_def by simp
  qed
  ultimately show "L |⊆| L1 |∪| L2'" by auto
qed
qed
qed
moreover have "l1 ∈ L1" using Cons(4) a_data.range_safe_subs by blast
ultimately show ?thesis using fold_subs[of xs "arange_safe (finsert l1 s) m1" "fset (L1 |∪|
L2')"] by fast
qed
qed
qed
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'"
using range_safe_update_subs[OF assms(1,2) _ _ _ assms(6,7,8)] unfolding arange_def

```

```
by (metis assms(3,4,5,9) a_data.range_def)
```

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)))"
  by pat_completeness auto
termination
  apply (relation "measure (λ(s,b). size (s))", auto)
  by (meson Suc_n_not_le_n leI size_list_estimation')

lemma write_sprefix: "sprefix m0 (snd (write cd m0))"
proof (induction arbitrary: m0 rule: write.induct)
  case (1 x m)
  then show ?case unfolding sprefix_def by (auto simp add:length_append_def)
next
  case (2 ds m)
  then have "prefix m0 (snd (fold_map write ds m0))"
  proof (induction ds arbitrary: m0)
    case Nil
    then show ?case by (simp add: prefix_def)
  next
    case (Cons a ds')
    then have IH: "∧m0. prefix m0 (snd (fold_map write ds' m0))" by simp
    show ?case
    proof (auto split:prod.split)
      fix m0 x1 x2 x1a x2a
      assume *: "fold_map write ds' x2 = (x1a, x2a)"
      and **: "write a m0 = (x1, x2)"

      from IH have "prefix x2 (snd (fold_map write ds' x2))" by simp
      then have "prefix x2 x2a" using * by simp
      moreover from ** have "prefix m0 x2" using Cons.premis[of a m0] sprefix_prefix by simp
      ultimately show "prefix m0 x2a" unfolding prefix_def by auto
    qed
  qed
  then show ?case unfolding sprefix_def prefix_def
  by (auto split:prod.split simp add:length_append_def)
qed

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))"
  using assms
proof (induction rule: dec_induct)
  case base
  then show ?case by blast
next
  case (step i)
  moreover from step(4)
  have "prefix (snd (fold_map write (take i ds) m0)) (snd (fold_map write (take (Suc i) ds) m0))"
  by (simp add: Suc_lessD assms fold_map_take_snd write_sprefix sprefix_prefix)
  ultimately show ?case unfolding loc_def prefix_def by auto
qed

lemma write_fold_map_sprefix:
  assumes "ds ≠ []"
  shows "sprefix m0 (snd (fold_map write ds m0))"
  using assms
proof (induction ds arbitrary: m0)
  case Nil
```

3 Stores and State

```

    then show ?case unfolding sprefix_def by auto
next
case (Cons a ds)
show ?case
proof (cases ds)
  case Nil
  then show ?thesis using write_sprefix[of m0 a] by (auto split:prod.split)
next
case *: (Cons a' list)
then have "sprefix (snd (write a m0)) (snd (fold_map write ds (snd (write a m0))))"
  using Cons by auto
then show ?thesis using write_sprefix[of m0 a] sprefix_trans by (auto split:prod.split)
qed
qed

```

```

lemma write_fold_map_mono:
  assumes "prefix ds' ds"
  shows "prefix (snd (fold_map write ds' m0)) (snd (fold_map write ds m0))"
  using assms
proof (induction ds' arbitrary: ds m0)
  case Nil
  then show ?case
proof (induction ds arbitrary: m0)
  case Nil
  then show ?case unfolding prefix_def by simp
next
  case (Cons a ds)
  then show ?case unfolding prefix_def apply (auto split:prod.split)
    by (metis append.assoc write_sprefix sndI sprefix_def)
qed
next
  case (Cons a ds')
  then show ?case
proof (induction ds arbitrary: m0)
  case Nil
  then show ?case unfolding prefix_def by simp
next
  case (Cons a ds)
  then show ?case unfolding prefix_def apply (auto split:prod.split)
    by (metis sndI)
qed
qed

```

```

lemma write_fold_map_smono:
  assumes "sprefix ds' ds"
  shows "sprefix (snd (fold_map write ds' m0)) (snd (fold_map write ds m0))"
  using assms
proof (induction ds' arbitrary: ds m0)
  case Nil
  then have "ds ≠ []" unfolding sprefix_def by blast
  then show ?case using write_fold_map_sprefix by auto
next
  case (Cons a ds')
  then show ?case
proof (induction ds arbitrary: m0)
  case Nil
  then show ?case unfolding sprefix_def by simp
next
  case (Cons a ds)
  then show ?case unfolding sprefix_def apply (auto split:prod.split) by (metis sndI)
qed
qed

```

```

lemma write_prefix_mono:

```

```

assumes "prefix ds' ds"
shows
  "prefix
    (butlast (snd (write (adata.Array ds') m0)))
    (butlast (snd (write (adata.Array ds) m0)))"
using assms apply (auto split: prod.split simp add:length_append_def)
using write_fold_map_mono
by (metis snd_conv)

lemma write_prefix_smono:
  assumes "sprefix ds' ds"
  shows
    "sprefix
      (butlast (snd (write (adata.Array ds') m0)))
      (butlast (snd (write (adata.Array ds) m0)))"
  using assms apply (auto split: prod.split simp add:length_append_def)
  using write_fold_map_smono by (metis snd_conv)

lemma write_length_inc: "length (snd (write cd m0)) > length m0"
  using write_sprefix[of m0 cd] unfolding sprefix_def by auto

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)))))"
  using assms
proof (induction ds arbitrary: m0 n)
  case Nil
  then show ?case by simp
next
  case (Cons a ds)
  show ?case
  proof (cases n)
    case 0
    then show ?thesis by (simp add:length_append_def split:prod.split)
  next
    case (Suc n')
    then have
      "fst (write (adata.Array (take (Suc n') ds)) (snd (write a m0)))
        = length (snd (write (ds ! n') (butlast (snd (write (adata.Array (take n' ds)) (snd (write a
m0)))))))"
      using Cons by auto
    then show ?thesis using Suc by (auto simp add:length_append_def split:prod.split)
  qed
qed

lemma butlast_write[simp]:
  "butlast (snd (write (adata.Array ds) m0)) = snd (fold_map write ds m0)"
  by (auto split:prod.split simp add:length_append_def)

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 -
  from assms have
    "prefix
      (snd (write (ds ! n) (snd (fold_map write (take n ds) m0))))
      (snd (fold_map write ds m0))"
  proof (induction ds arbitrary: m0 n)
    case Nil
    then show ?case by simp
  
```

```

next
  case (Cons a xs)
  show ?case
  proof (cases n)
  case 0
  then show ?thesis apply (auto split:prod.split)
  by (metis fold_map_prefix write_sprefix snd_eqD sprefix_prefix)
next
  case (Suc n')
  then have "n' < length xs" using Cons(2) by simp
  then have
    "prefix
     (snd (write (xs ! n') (snd (fold_map write (take n' xs) (snd (write a m0))))))
     (snd (fold_map write xs (snd (write a m0))))"
    using Cons(1)[of "n'" "(snd (write a m0))"] by simp
  moreover have "(a # xs) ! n = (xs ! n'" using Suc by simp
  moreover have
    "(snd (fold_map write (take n' xs) (snd (write a m0))))
     = (snd (fold_map write (take n (a # xs)) m0))"
    using Suc(1) by (auto split:prod.split)
  moreover have
    "(snd (fold_map write (a # xs) m0)) = (snd (fold_map write xs (snd (write a m0))))"
    by (auto split:prod.split)
  ultimately show ?thesis by simp
qed
qed
then show ?thesis apply (auto simp add:length_append_def split:prod.split)
  by (smt (verit) Nil_is_append_conv append_assoc not_Cons_self2 prefix_def sprefix_def)
qed

lemma write_length_suc: "length (snd (write ds m)) = Suc (fst (write ds m))"
  by (cases ds) (auto simp add:length_append_def split:prod.split)

lemma write_length_suc2:
  assumes "write ds m0 = (l, m)"
  shows "l = length m - 1"
  using assms write_length_suc[of ds m0] by simp

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)"
  using assms
proof -
  have "n < length ds" using assms
  by (simp add: fold_map_length)

  have "fst (fold_map write ds m) ! n = fst (write (ds ! n) (snd (fold_map write (take n ds) m)))"
  using fold_map_take_fst[OF assms] by simp
  also have "Suc (...) = length (snd (write (ds ! n) (snd (fold_map write (take n ds) m))))"
  by (simp add: write_length_suc)
  also have "... < length (snd (write (adata.Array ds) m))"
  using write_sprefix_take[OF <n < length ds>, of m] unfolding sprefix_def by auto
  also have "... = Suc (fst (write (adata.Array ds) m))"
  by (auto split:prod.split simp add: length_append_def)
  finally show ?thesis by blast
qed

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))"

```

```

m0))"
proof -
  obtain xs m
  where *: "snd (write (Array ds) m0) = m @ [mdata.Array xs]"
  and **: "xs = fst (fold_map write ds m0)"
  and ***: "fst (write (Array ds) m0) = length m"
  apply (auto simp add:length_append_def split:prod.split prod.split)
  by (simp add: case_prod_beta')

from ** have "∀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))"
  using fold_map_take_fst write_fold_map_less unfolding prefix_def
  by (metis fold_map_length write_sprefix_take sprefix_def)
moreover from ** have "(length xs = length ds)" using Utils.fold_map_length by metis
moreover from * ***
have "snd (write (Array ds) m0) $ fst (write (Array ds) m0) = Some (mdata.Array xs)"
  by auto
ultimately show ?thesis using that by blast
qed

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"
  using assms
proof -
  from assms obtain "ds" where 2: "cd = Array ds" by (case_tac cd, auto simp add: length_append_def)
  then have "snd (write (Array ds) m) $ fst (write (Array ds) m) = Some (mdata.Array xs)"
  using assms by auto
  moreover have "i < length xs" using assms unfolding nth_safe_def by (simp split:if_split_asm)
  ultimately show ?thesis using write_obtain[of ds m]
  by (metis "2" assms(1,3) mdata.inject(2) nth_safe_def option.inject split_pairs2)
qed

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 (rule allI, rule impI, rule impI)
  fix l
  assume *: "length (snd (fold_map write (take n ds) m0)) ≤ l"
  and **: "l < length (snd (write (ds ! n) (snd (fold_map write (take n ds) m0))))"
  from * have "l ≥ length m0"
  by (meson dual_order.trans fold_map_geq write_length_inc order.strict_implies_order)
  moreover from assms(2) ** have "l < length (snd (write (adata.Array ds) m0))"
  using write_sprefix_take[OF assms(1), of m0] unfolding sprefix_def by fastforce
  moreover have "l ≠ fst (write (adata.Array ds) m0)"
  proof -
    from write_length_suc have
      "length (snd (write (ds ! n) (snd (fold_map write (take n ds) m0))))
      = Suc (fst (write (ds ! n) (snd (fold_map write (take n ds) m0))))"
    by auto
    also from assms(3) have "... ≤ fst (write (adata.Array ds) m0)" using assms(2) by auto
    finally show ?thesis using ** by simp
  qed
  ultimately show "l ∉ finsert (fst (write (adata.Array ds) m0)) s" using assms(4) by simp

```

qed

```

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 (rule allI, rule impI, rule impI)
  fix l
  assume *: "length (butlast (snd (write (adata.Array (take n ds)) m0))) ≤ l"
    and **: "l < length (snd (write (ds ! n) (butlast (snd (write (adata.Array (take n ds)) m0)))))"
  from * have "l ≥ length m0"
    by (metis diff_Suc_1 dual_order.trans length_butlast less_Suc_eq_le write_length_inc
write_length_suc)
  moreover from ** have "l < length (snd (write (adata.Array ds) m0))"
  proof -
    from ** have "l < length (snd (fold_map write (take (Suc n) ds) m0))"
      using fold_map_take_snd assms(1) by (metis butlast_write)
    then show ?thesis
      by (metis (no_types, lifting) assms(1) dual_order.strict_trans fold_map_take_snd
write_sprefix_take sprefix_length)
  qed
  ultimately have "l |∈| s" using assms(2) by blast
  moreover have "l ≠ fst (write (adata.Array ds) m0)" using assms(3,4)
    by (metis "***" butlast_write write_length_suc not_less_eq)
  ultimately show "l |∈| finsert (fst (write (adata.Array ds) m0)) s" by blast
qed

```

```

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'"
  using assms
proof (induction "is" arbitrary: m l m' ns l' m'' ds i rule: list_nonempty_induct)
  case (single i0)

  from single(1,2)
  have 1: "ns!i = fst (write (ds!i) (snd (fold_map write (take i ds) m)))"
    and 2: "prefix (snd (write (ds!i) (snd (fold_map write (take i ds) m)))) (snd (write (Array ds) m))"
  using write_obtain[of ds m] nth_safe_length single.prem(3)
  by (metis fstI mdata.inject(2) option.inject sndI)+

  have
    3: "snd (write (ds!i) (snd (fold_map write (take i ds) m)))
      = snd (fold_map write (take (Suc i) ds) m)"
  by (metis fold_map_take_snd fst_conv mdata.inject(2) write_obtain nth_safe_length
option.inject single.prem(1,2,3) snd_conv)

  then have "prefix (snd (fold_map write (take (Suc i) ds) m)) m'" using single(5,1) 2 prefix_def
  by (metis append.assoc fst_conv fst_swap swap_simp)

  moreover have "(snd (fold_map write (take (Suc i) ds) m)) $ l' ≠ None" using 1 3 single.prem(3)
  by (metis less_Suc_eq write_length_suc nth_safe_def option.distinct(1) option.inject)

  ultimately have "(snd (fold_map write (take (Suc i) ds) m)) $ l' = m' $ l'"
  using single nth_safe_prefix by fastforce
  then show ?case by (auto simp add:case_memory_def)

```

```

next
case (cons i0 is0)
from cons(1) obtain i0' is0' where is0_def: "is0 = i0' # is0'"
  using list.exhaust by auto
with cons(6) obtain ns1 i1 l1
  where l1: "m'' $ l' = Some (mdata.Array ns1)"
  and l2: "to_nat i0 = Some i1"
  and l3: "ns1 $ i1 = Some l1"
  and l4: "marray_lookup m'' is0 l1 = Some (lx, nsx, ix)"
  using marray_lookup_obtain_multi[of m'' i0 i0' "is0'" l' "(lx, nsx, ix)"] by blast

from cons have 0: "∀n < length ns. ns!n < fst (write (Array ds) m)
  ∧ ns!n = fst (write (ds!n) (snd (fold_map write (take n ds) m)))
  ∧ prefix (snd (write (ds!n) (snd (fold_map write (take n ds) m)))) (snd (write (Array ds)
m))"
  using write_obtain[of ds m]
  by (metis (no_types, lifting) fst_conv mdata.inject(2) option.inject snd_conv)
with cons have 1: "ns!i = fst (write (ds!i) (snd (fold_map write (take i ds) m)))"
  by (metis nth_safe_length)

have "fst (write (ds!i) (snd (fold_map write (take i ds) m))) = l'"
  by (metis "1" cons.prems(3) nth_safe_def nth_safe_length option.sel)
then have
  "snd (write (ds!i) (snd (fold_map write (take i ds) m)))
  $ fst (write (ds!i) (snd (fold_map write (take i ds) m))) = Some (mdata.Array ns1)"
  using l1 cons(7)
  by (smt (verit) "0" cons.prems(1,3) lessI write_length_suc nth_safe_length nth_safe_prefix
    nth_safe_some snd_conv)
then obtain "ds'" where 2: "ds!i = adata.Array ds'"
  apply (case_tac "ds!i") by (auto simp add:length_append_def)

have 3:
  "snd (write (ds ! i) (snd (fold_map write (take i ds) m)))
  $ fst (write (ds ! i) (snd (fold_map write (take i ds) m)))
  = Some (mdata.Array ns1)"
  by (metis (no_types, lifting) "0" "2" cons.prems(1,3,5) l1 write_obtain nth_safe_length
    nth_safe_prefix nth_safe_some option.inject snd_eqD)

have 4:
  "marray_lookup
  (snd (fold_map write (take (Suc i1) ds') (snd (fold_map write (take i ds) m)))) is0 l1
  = marray_lookup m'' is0 l1"
proof (rule cons(2)[
  of
  -
  "(snd (fold_map write (take i ds) m))"
  "fst (write (ds!i) (snd (fold_map write (take i ds) m)))"
  "snd (write (ds!i) (snd (fold_map write (take i ds) m)))",
  OF _ _ 13 14])
  from 1 2 show
    "write (adata.Array ds') (snd (fold_map write (take i ds) m))
    = (fst (write (ds ! i) (snd (fold_map write (take i ds) m))),
      snd (write (ds ! i) (snd (fold_map write (take i ds) m))))"
    by simp
next
from 3 show
  "snd (write (ds ! i) (snd (fold_map write (take i ds) m)))
  $ fst (write (ds ! i) (snd (fold_map write (take i ds) m)))
  = Some (mdata.Array ns1)" by simp
next
have "prefix (snd (write (ds ! i) (snd (fold_map write (take i ds) m)))) m'" using cons(3) 0
  by (metis cons.prems(3) nth_safe_length snd_conv)
moreover have "prefix m' m'" using cons(7) by simp
ultimately show "prefix (snd (write (ds ! i) (snd (fold_map write (take i ds) m)))) m'"

```

```

    unfolding prefix_def by auto
qed

have "marray_lookup m'' (i0 # is0) l' = marray_lookup m'' is0 l1"
  using l1 l2 l3 is0_def by (auto simp add:case_memory_def)
also from 4 have
  "... =
  marray_lookup
    (snd (fold_map write (take (Suc i1) ds') (snd (fold_map write (take i ds) m)))) is0 l1" ..
also have "... = marray_lookup (snd (fold_map write (take (Suc i) ds) m)) (i0 # is0) l'"
proof -
have
  "prefix
    (snd (fold_map write (take (Suc i1) ds') (snd (fold_map write (take i ds) m))))
    (snd (fold_map write (take (Suc i) ds) m))"
  by (metis (no_types, lifting) "2" "3" l3 cons.prems(1,2,3) fold_map_take_snd
    fst_conv mdata.inject(2) write_obtain_nth_safe_length option.inject snd_conv)
then have
  "marray_lookup (snd (fold_map write (take (Suc i1) ds') (snd (fold_map write (take i ds) m)))) is0
11
  =marray_lookup (snd (fold_map write (take (Suc i) ds) m)) is0 l1"
  by (metis calculation cons.prems(4) marray_lookup_prefix)
moreover have
  "marray_lookup (snd (fold_map write (take (Suc i) ds) m)) is0 l1
  = marray_lookup (snd (fold_map write (take (Suc i) ds) m)) (i0 # is0) l'"
proof -
  have "(snd (fold_map write (take (Suc i) ds) m)) $ l' = Some (mdata.Array ns1)" using l1
    by (metis "3" cons.prems(1,2,3) fold_map_take_snd fst_conv mdata.inject(2)
      write_obtain_nth_safe_length nth_safe_some option.inject snd_conv)
  then show ?thesis using l2 l3 is0_def by (auto simp add: case_memory_def)
qed
ultimately show ?thesis by simp
qed
finally show ?case by simp
qed

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'"
using assms
proof (induction "is" arbitrary: m l m' ns l' m'' ds i L)
  case Nil
  then show ?case by simp
next
  case (Cons i0 "is0")

  from Cons(5) obtain ns1 i1 l1 L'
    where l1: "m'' $ l' = Some (mdata.Array ns1)"
    and l2: "to_nat i0 = Some i1"
    and l3: "ns1 $ i1 = Some l1"
    and l4: "locations m'' is0 l1 = Some L'"
    and l5: "L = (finsert l' L)"
  using locations_obtain[of m'' i0 is0 l' L] by blast

  from Cons have
  0: "∀n < length ns. ns!n < fst (write (Array ds) m)
    ∧ ns!n = fst (write (ds!n) (snd (fold_map write (take n ds) m)))
    ∧ prefix (snd (write (ds!n) (snd (fold_map write (take n ds) m)))) (snd (write (Array ds) m))"
  using write_obtain[of ds m]
  by (metis (no_types, lifting) fst_conv mdata.inject(2) option.inject snd_conv)

```

```

with Cons have 1: "ns!i = fst (write (ds!i) (snd (fold_map write (take i ds) m)))"
  by (metis nth_safe_length)

have "fst (write (ds!i) (snd (fold_map write (take i ds) m))) = 1'"
  by (metis "1" Cons.prem3) nth_safe_def nth_safe_length option.sel)
then have
  "snd (write (ds!i) (snd (fold_map write (take i ds) m)))
  $ fst (write (ds!i) (snd (fold_map write (take i ds) m))) = Some (mdata.Array ns1)"
  using l1 Cons
  by (smt (verit) "0" Cons lessI write_length_suc nth_safe_length nth_safe_prefix nth_safe_some
    snd_conv)
then obtain "ds'" where 2: "ds!i = adata.Array ds'"
  apply (case_tac "ds!i") by (auto simp add:length_append_def)

have
  3: "snd (write (ds ! i) (snd (fold_map write (take i ds) m)))
    $ fst (write (ds ! i) (snd (fold_map write (take i ds) m)))
    = Some (mdata.Array ns1)"
  by (metis (no_types, lifting) "0" "2" Cons(2,4,6) l1 write_obtain nth_safe_length
    nth_safe_prefix nth_safe_some option.inject snd_eqD)

have 4:
  "locations (snd (fold_map write (take (Suc i1) ds') (snd (fold_map write (take i ds) m)))) is0 l1
  = locations m'' is0 l1"
proof (rule Cons(1)[
  of
  -
  "(snd (fold_map write (take i ds) m))"
  "fst (write (ds!i) (snd (fold_map write (take i ds) m)))"
  "snd (write (ds!i) (snd (fold_map write (take i ds) m)))",
  OF _ _ 13 14])
  from 1 2 show
    "write (adata.Array ds') (snd (fold_map write (take i ds) m))
    = (fst (write (ds ! i) (snd (fold_map write (take i ds) m))),
      snd (write (ds ! i) (snd (fold_map write (take i ds) m))))"
    by simp
next
  from 3 show
    "snd (write (ds ! i) (snd (fold_map write (take i ds) m)))
    $ fst (write (ds ! i) (snd (fold_map write (take i ds) m)))
    = Some (mdata.Array ns1)" by simp
next
  have "prefix (snd (write (ds ! i) (snd (fold_map write (take i ds) m)))) m'" using Cons(3) 0
    by (metis Cons(2,4) nth_safe_length snd_conv)
  moreover have "prefix m' m'" using Cons by simp
  ultimately show "prefix (snd (write (ds ! i) (snd (fold_map write (take i ds) m)))) m'"
    unfolding prefix_def by auto
qed

have "locations m'' (i0 # is0) l' = Some (finsert l' (the (locations m'' is0 l1)))"
  using l1 l2 l3 l4 Cons by (auto simp add:case_memory_def)
also from 4 have
  "locations m'' is0 l1
  = locations (snd (fold_map write (take (Suc i1) ds') (snd (fold_map write (take i ds) m)))) is0 l1"
..
also have
  "Some (
    finsert
    l'
    (the (locations (snd (fold_map write (take (Suc i1) ds') (snd (fold_map write (take i ds) m))))
is0 l1)))
  = locations (snd (fold_map write (take (Suc i) ds) m)) (i0 # is0) l'"
proof -
  have

```

```

"prefix
  (snd (fold_map write (take (Suc i1) ds') (snd (fold_map write (take i ds) m))))
  (snd (fold_map write (take (Suc i) ds) m))"
by (metis "2" "3" Cons.prems(1,2,3) l3 fold_map_take_snd fst_conv mdata.inject(2)
  write_obtain_nth_safe_length option.inject_snd_conv)
then have
  "locations (snd (fold_map write (take (Suc i1) ds') (snd (fold_map write (take i ds) m)))) is0 l1
  = locations (snd (fold_map write (take (Suc i) ds) m)) is0 l1"
by (metis "4" l4 locations_prefix_locations)
moreover have
  "Some (finsert l' (the (locations (snd (fold_map write (take (Suc i) ds) m)) is0 l1)))
  = locations (snd (fold_map write (take (Suc i) ds) m)) (i0 # is0) l'"
proof -
  have "(snd (fold_map write (take (Suc i) ds) m)) $ l' = Some (mdata.Array ns1)" using l1
  by (metis "3" Cons(2,3,4) fold_map_take_snd fst_conv mdata.inject(2) write_obtain
    nth_safe_length nth_safe_some option.inject_snd_conv)
  then show ?thesis using l2 l3 l4 apply (auto simp add: case_memory_def)
  apply (case_tac "locations (snd (fold_map write (take (Suc i) ds) m)) is0 l1", auto)
  by (metis "4" calculation option.distinct(1))
qed
ultimately show ?thesis by simp
qed
finally show ?case by simp
qed

```

3.18 Memory Init and Lookup (Memory)

```

lemma marray_lookup_write_less:
  assumes "is  $\neq$  []"
  and "write cd m = (l, m'"
  and "marray_lookup m' is l = Some (lx, nsx, ix)"
  and "nsx $ ix = Some l'"
  shows "l' < l"
  using assms
proof (induction "is" arbitrary: m l m' cd rule: list_nonempty_induct)
  case (single i0)
  then have "m' $ lx = Some (mdata.Array nsx)" and "to_nat i0 = Some ix" and "lx = l"
  using marray_lookup_obtain_single[OF single(2)] by auto
  then show ?case using single.prems(1,3) write_array_less by blast
next
  case (cons i0 is0)

  from cons(1) obtain i0' is0'
  where is0_def: "is0 = i0' # is0'"
  using list.exhaust by auto
  with cons(4) obtain ns1 i1 l1
  where l1: "m' $ l = Some (mdata.Array ns1)"
  and l2: "to_nat i0 = Some i1"
  and l3: "ns1 $ i1 = Some l1"
  and l4: "marray_lookup m' is0 l1 = Some (lx, nsx, ix)"
  using marray_lookup_obtain_multi[of m' i0 i0' "is0'" l "(lx, nsx, ix)"] by blast

  from cons(3) l1 obtain "ds" where 2: "cd = Array ds" by (case_tac cd, auto simp add:
  length_append_def)

  have "l1 < l" by (meson cons.prems(1) l1 l3 write_array_less)
  moreover have "l' < l1"

proof (rule cons(2)[OF _ _ cons(5)])
  from cons have
    0: " $\forall n < \text{length } ns1. ns1!n < \text{fst } (\text{write } (\text{Array } ds) m)$ "
     $\wedge ns1!n = \text{fst } (\text{write } (ds!n) (\text{snd } (\text{fold\_map } \text{write } (\text{take } n \text{ ds}) m)))$ 
     $\wedge \text{prefix } (\text{snd } (\text{write } (ds!n) (\text{snd } (\text{fold\_map } \text{write } (\text{take } n \text{ ds}) m)))) (\text{snd } (\text{write } (\text{Array } ds) m))"$ 
  using write_obtain[of ds m]

```

```

    by (smt (verit, ccfv_SIG) "2" l1 mdata.inject(2) option.inject split_pairs2)
  then show
    "write (ds!i1) (snd (fold_map write (take i1 ds) m))
    = (l1, (snd (fold_map write (take (Suc i1) ds) m)))"
    by (metis "2" l1 cons.prem1 fold_map_take_snd l3 mdata.inject(2) write_obtain
        nth_safe_def nth_safe_length option.inject split_pairs2)
  next
    from 14 show "marray_lookup (snd (fold_map write (take (Suc i1) ds) m)) is0 l1 = Some (lx, nsx,
ix)"
    using marray_lookup_write_take[OF cons(1) _ l1 l3 l4 ] cons(3) 2 by auto
  qed
  ultimately show ?case by simp
qed

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"
  using assms
proof (induction xs arbitrary:L l cd m m')
  case Nil
  then show ?case by simp
next
  case (Cons i xs')
  then show ?case
proof (cases xs')
  case Nil
  then show ?thesis
proof (cases cd)
  case (Value x1)
  then show ?thesis
  using Cons(2,3,4) Nil by (auto simp add: length_append_def case_memory_def)
next
  case (Array ca)
  then obtain ns m''
    where "fold_map write ca m = (ns, m'')"
      and "l = length m'"
      and "m' = m'' @ [mdata.Array ns]"
    using Cons(2,3,4) Nil apply (case_tac "fold_map write ca m")
    by (auto simp add: length_append_def case_memory_def)
  then have *: "m'$l = Some (mdata.Array ns)" using Cons by simp
  moreover from * obtain i''
    where ***: "to_nat i = Some i'"
      and "(l1, xs1, i1) = (l, ns, i'')"
    using Cons Nil apply (auto simp add: length_append_def case_memory_def)
    by (case_tac "vtype_class.to_nat i") auto
  then have **: "ns$i'' =Some l2" using Cons by simp
  moreover have "locations m' (i # xs') l =Some {l1}"
    using Cons(5) Nil * ** *** by (auto simp add: length_append_def case_memory_def)
  ultimately show ?thesis using Cons.prem4 write_array_less Cons(2) by fastforce
qed
next
  case c2: (Cons i' is')
  then show ?thesis
proof (cases cd)
  case (Value x1)
  then show ?thesis
  using Cons(2,3,4) c2 Nil by (auto simp add: length_append_def case_memory_def)
next
  case (Array ca)
  then obtain ns m''
    where 0: "fold_map write ca m = (ns, m'')"

```

```

    and "l = length m'"
    and "m' = m' @ [mdata.Array ns]"
using Cons(2,3,4) Nil apply (case_tac "fold_map write ca m")
by (auto simp add: length_append_def case_memory_def)

then obtain i'' l'
  where 1: "m' $ l = Some (mdata.Array ns)"
    and 2: "to_nat i = Some i'"
    and 3: "ns $ i'' = Some l'"
    and 4: "marray_lookup m' xs' l' = Some (l1, xs1, i1)"
  using marray_lookup_obtain_multi[of m' i i' is' l "(l1, xs1, i1)"] Cons(3) c2
  by (metis mdata.inject(2) nth_append_length nth_safe_length nth_safe_some option.inject)

from 1 2 3 4 obtain L'
where 5: "locations m' (i'#is') l' = Some L'"
  and 6: "L = (finsert l L')"
  using locations_obtain[OF Cons(5)] c2 by force

have "i'' < length (fst (fold_map write ca m))"
  by (simp add: "3" <fold_map write ca m = (ns, m'')> nth_safe_length)
then have
  7: "fst (fold_map write ca m) ! i''
  = fst (write (ca ! i'') (snd (fold_map write (take i'' ca) m)))"
  using fold_map_take_fst by metis

have "i'' < length ca"
  by (metis <i'' < length (fst (fold_map write ca m))> fold_map_length)
then have
  8: "snd (fold_map write (take (Suc i'') ca) m)
  = snd (write (ca ! i'') (snd (fold_map write (take i'' ca) m)))"
  using fold_map_take_snd by metis

obtain mx lx
  where 12: "fold_map write (take i'' ca) m = (lx, mx)"
    and 13: "write (ca ! i'') mx
    = (fst (fold_map write ca m) ! i'', snd (fold_map write (take (Suc i'') ca) m))"
  using 7 8 by fastforce

have "(fst (fold_map write ca m) ! i'') = l'"
  by (metis "3" 0 fst_conv nth_safe_length nth_safe_some option.inject)
then have
  "marray_lookup (snd (fold_map write (take (Suc i'') ca) m)) xs' (fst (fold_map write ca m) !
i'')
  = marray_lookup m' xs' l'"
  using c2 marray_lookup_write_take 1 "2" 3 "4" Array Cons.prems(1,2) 'i'' < length ca'
  by blast
then have
  9: "marray_lookup (snd (fold_map write (take (Suc i'') ca) m)) xs' (fst (fold_map write ca m) !
i'')
  = Some (l1, xs1, i1)" using 4 by simp

have "fst (fold_map write ca m) ! i'' = l'"
  by (metis "3" <fold_map write ca m = (ns, m'')> fst_conv nth_safe_length nth_safe_some
option.inject)
then have
  "locations (snd (fold_map write (take (Suc i'') ca) m)) xs' (fst (fold_map write ca m) ! i'')
  = locations m' (i'#is') l'"
  using c2 locations_lookup_write_take "1" "3" Array Cons.prems(1) 5 by blast
then have
  10: "locations (snd (fold_map write (take (Suc i'') ca) m)) xs' (fst (fold_map write ca m) !
i'')
  = Some L'" using 5 by simp

have "12 |∈| L'" using Cons(1)[OF 13 9 Cons(4) 10] by simp

```

```

    moreover from marray_lookup_write_less have "l2 ≠ l"
      using Cons.prem1,2 assms(3) by blast
    ultimately show ?thesis using l2 l3 l6 by blast
qed
qed
qed

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"
  using assms
proof (induction xs arbitrary: m l m' cd x m' rule: list_nonempty_induct)
  case (single i0)
  from single(2) obtain ds i
    where "cd = adata.Array ds"
      and l2: "to_nat i0 = Some i"
      and l3: "ds $ i = Some x"
  apply (cases cd,auto)
  apply (cases "to_nat i0",auto)
  by (case_tac "x2$a",auto)
  then show ?case using single.prem1,3
  apply (auto simp add:case_memory_def)
  apply (cases "m' $ l",auto)
  apply (metis fst_conv write_obtain_nth_safe_prefix option.distinct(1) single.prem1,3 snd_conv)
  apply (case_tac a,auto simp add:nth_safe_def length_append_def split:if_split_asm prod.split_asm)
  apply (metis length_append_singleton lessI mdata.distinct(1) nth_append_left nth_append_length
prefix_def)
  by (metis fold_map_length fst_eqD length_append_singleton mdata.inject(2) not_less_eq
nth_append_left nth_append_length prefix_def verit_comp_simplify1(1))
next
  case (cons i0 is0)

  from cons(4) obtain ds i cd'
    where a1: "cd = adata.Array ds"
      and a2: "to_nat i0 = Some i"
      and a3: "ds $ i = Some cd'"
      and a4: "alookup is0 cd' = Some x"
  apply (cases cd,auto)
  apply (cases "to_nat i0",auto)
  by (case_tac "x2$a",auto)

  then obtain ns
    where b1: "m' $ l = Some (mdata.Array ns)"
      and b2: "length ds = length ns"
      and b3: "∀ n < length ns. ns!n < fst (write (Array ds) m)
        ∧ ns!n = fst (write (ds!n) (snd (fold_map write (take n ds) m)))
        ∧ prefix (snd (write (ds!n) (snd (fold_map write (take n ds) m)))) (snd (write (Array ds)
m))"
  using write_obtain[of ds m]
  using cons(4)
  by (metis cons.prem1 fstI write_obtain snd_eqD)
  moreover from a3 have "i < length ds" unfolding nth_safe_def by (simp split:if_split_asm)
  ultimately have *: "ns!i = fst (write (ds!i) (snd (fold_map write (take i ds) m)))"
  by (simp add: <length ds = length ns>)

  have
    "∃ lz xsz iz z. marray_lookup m' is0 (fst (write (ds!i) (snd (fold_map write (take i ds) m))))
    = Some (lz, xsz, iz) ∧ xsz $ iz = Some z"
  proof (rule cons(2)[OF _ a4])
    from * show
      "write cd' (snd (fold_map write (take i ds) m)) =

```

```

      (fst (write (ds ! i) (snd (fold_map write (take i ds) m))),
      snd (write (ds ! i) (snd (fold_map write (take i ds) m))))"
    using <i < length ds> a3 by fastforce
  next
  have "prefix (snd (write (ds ! i) (snd (fold_map write (take i ds) m)))) m'"
    by (metis a1 b2 b3 <i < length ds> cons.prems(1) snd_conv)
  then show "prefix (snd (write (ds ! i) (snd (fold_map write (take i ds) m)))) m'"
    using cons(5) prefix_trans by blast
qed
then obtain lz xsz iz z
  where
    "marray_lookup m'' is0 (fst (write (ds!i) (snd (fold_map write (take i ds) m))))
    = Some (lz, xsz, iz)"
    and "xsz $ iz = Some z"
  by blast
moreover from b1 have "m''$1 = Some (mdata.Array ns)"
  by (meson cons.prems(3) nth_safe_prefix)
ultimately show ?case using * a2 b2 <i < length ds>
  by (cases is0,auto simp add:case_memory_def)
qed

lemma mlookup_some:
  assumes "write cd m = (l, m')"
    and "alookup xs cd = Some x"
  shows "∃y. mlookup m' xs l = Some y"
proof (cases xs)
  case Nil
  then show ?thesis by simp
next
  case (Cons a list)
  moreover obtain lz xsz iz z
    where "marray_lookup m' (a # list) l = Some (lz, xsz, iz)"
    and "xsz $ iz = Some z"
  using write_lookup_some[of "a # list", OF _ assms(1), of x m']
  using assms(2) Cons by blast
  ultimately show ?thesis by simp
qed

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 (cases xs)
  case Nil
  then show ?thesis
    using assms(3) by auto
next
  case (Cons a list)
  then obtain l1' xs1 i1 l2
    where "marray_lookup m' xs l = Some (l1', xs1, i1)"
    and "xs1 $ i1 = Some l2"
    and "l1 = l2"
  using mlookup_obtain_empty1[of m' a list l l1] using assms(2) by metis
  then have "l2 |∈| L" using write_marray_lookup_locations[OF assms(1) _ _ assms(3)] by blast
  then show ?thesis using 'l1 = l2' by simp
qed

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"
  using assms

```

```

proof (induction xs arbitrary: m l m' cd x m'')
  case Nil
  then show ?case by simp
next
case (Cons i0 "is")
from Cons(3) obtain ds i cd'
  where a1: "cd = adata.Array ds"
  and a2: "to_nat i0 = Some i"
  and a3: "ds $ i = Some cd'"
  and a4: "alookup is cd' = Some x"
apply (cases cd,auto)
apply (cases "to_nat i0",auto)
by (case_tac "x2$a",auto)

then obtain ns
  where b1: "m' $ l = Some (mdata.Array ns)"
  and b2: "length ds = length ns"
  and b3: "∀n < length ns. ns!n < fst (write (Array ds) m)
    ∧ ns!n = fst (write (ds!n) (snd (fold_map write (take n ds) m)))
    ∧ prefix (snd (write (ds!n) (snd (fold_map write (take n ds) m)))) (snd (write (Array ds)
m))"
  using write_obtain[of ds m]
  using Cons
  by (metis Cons(2) fstI write_obtain snd_eqD)
moreover from a3 have "i < length ds" unfolding nth_safe_def by (simp split:if_split_asm)
ultimately have *: "ns!i = fst (write (ds!i) (snd (fold_map write (take i ds) m)))"
  by (simp add: <length ds = length ns>)

have "∃y. locations m'' is (fst (write (ds!i) (snd (fold_map write (take i ds) m)))) = Some y"
proof (rule Cons(1)[OF _ a4])
  from * show
    "write cd' (snd (fold_map write (take i ds) m)) =
      (fst (write (ds ! i) (snd (fold_map write (take i ds) m))),
        snd (write (ds ! i) (snd (fold_map write (take i ds) m))))"
    using <i < length ds> a3 by fastforce
next
have "prefix (snd (write (ds ! i) (snd (fold_map write (take i ds) m)))) m'"
  by (metis b2 b3 a1 <i < length ds> Cons.prem1 snd_conv)
then show "prefix (snd (write (ds ! i) (snd (fold_map write (take i ds) m)))) m'"
  using Cons prefix_trans by blast
qed
then obtain y
  where "locations m'' is (fst (write (ds!i) (snd (fold_map write (take i ds) m)))) = Some y"
  by blast
moreover from b1 have "m''$l = Some (mdata.Array ns)"
  by (meson Cons nth_safe_prefix)
ultimately show ?case using * a2 b2 <i < length ds>
  by (cases "is",auto simp add:case_memory_def)
qed

```

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 -
  from assms write_obtain[of ds m0] obtain xs
  where *: "m $ l = Some (mdata.Array xs)"
  and "length xs = length ds"
  and **: "∀n < length xs.

```

```

xs ! n < fst (write (adata.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)))) m"
by auto
moreover from assms(2) * have
"fold
  (λx y. y ≧ (λy'. (arange_safe (finsert l s) m x) ≧ (λl. Some (l |∪| y'))))
  xs
  (Some {||})
  = Some L"
by (auto simp add:case_memory_def split:if_split_asm)
ultimately have
"x = l ∨ (∃n L'. n < length xs ∧ arange_safe (finsert l s) m (xs ! n) = Some L' ∧ x |∈| L'"
  using fold_union_in[of "arange_safe (finsert l s) m"] assms(3) by blast
then show ?thesis
proof
  assume "x = l"
  then show ?thesis by simp
next
  assume "∃n L'. n < length xs ∧ arange_safe (finsert l s) m (xs ! n) = Some L' ∧ x |∈| L'"
  then obtain n L'
    where "n < length xs"
      and ***: "arange_safe (finsert l s) m (xs ! n) = Some L'"
      and "x |∈| L'" by blast
  moreover from *** have "arange_safe s m (xs ! n) = Some L'"
    using a_data.range_safe_subset_same by blast
  ultimately show ?thesis using ** by (metis <length xs = length ds>)
qed
qed

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)))"
  using assms
proof (induction cd arbitrary: m0 s)
  case (Value x)
  then show ?case
    by (auto simp add: s_disj_fs_def pred_some_def length_append_def case_memory_def loc_def)
next
  case (Array ds)
  from write_obtain obtain xs
    where xs1: "snd (write (Array ds) m0) $ fst (write (Array ds) m0) = Some (mdata.Array xs)"
      and xs2: "length xs = length ds"
      and xs3: "∀n < length xs. xs!n < fst (write (Array ds) m0) ∧
        xs!n = fst (write (ds!n) (snd (fold_map write (take n ds) m0)))"
  by metis
  moreover have "¬ fst (write (Array ds) m0) |∈| s" using Array(2)
  by (metis less_Suc_eq_le write_length_inc write_length_suc nth_safe_length xs1)
  ultimately have
    "arange_safe s
      (snd (write (adata.Array ds) m0))
      (fst (write (adata.Array ds) m0))
    = fold (λx y.
      y ≧ (λy'. (arange_safe (finsert (fst (write (Array ds) m0)) s) (snd (write (Array ds) m0)) x)
        ≧ (λl. Some (l |∪| y'))))
      xs (Some {|fst (write (Array ds) m0)|})" (is "_ = fold ?f xs (Some {|fst (write (Array ds)
m0)|})")
  by (simp add:case_memory_def)
  moreover have "s_disj_fs (loc m0) (fold ?f xs (Some {|fst (write (Array ds) m0)|}))"
  proof (rule take_all1[of xs])
    show "∀n ≤ length xs. s_disj_fs (loc m0) (fold ?f (take n xs) (Some {|fst (write (Array ds)
m0)|}))"
  proof (rule allI, rule impI)
    fix n

```

```

assume "n ≤ length xs"
then show "s_disj_fs (loc m0) (fold ?f (take n xs) (Some {fst (write (Array ds) m0)}))"
proof (induction n)
  case 0
  then show ?case
    using write_fold_map_sprefix[of ds m0, THEN sprefix_length]
    by (fastforce simp add:s_disj_fs_def pred_some_def length_append_def loc_def
split:prod.split)
  next
  case (Suc n)

  from Suc(2) have "n < length xs" by auto
  then have "n < length ds" using xs2 by simp

  from Suc(2) have
    "fold ?f (take (Suc n) xs) (Some {fst (write (Array ds) m0)})
  = ?f (xs!n) (fold ?f (take n xs) (Some {fst (write (Array ds) m0)}))"
  by (simp add: fold_take)
  moreover have "s_disj_fs (loc m0) (fold ?f (take n xs) (Some {fst (write (Array ds) m0)}))"
  using Suc by simp
  moreover have "∧s. s_disj_fs (loc m0) s ⇒ s_disj_fs (loc m0) (?f (xs!n) s)"
  proof -
    fix x assume "s_disj_fs (loc m0) x"
    moreover have
      "s_disj_fs
      (loc m0)
      (arange_safe
      (finsert (fst (write (adata.Array ds) m0)) s)
      (snd (write (adata.Array ds) m0))
      (xs ! n))"
    proof -
      have "(ds!n) ∈ set ds" using Suc(2) by (simp add: xs2)
      moreover have "∀l ≥ length (snd (fold_map write (take n ds) m0)).
        1 < length (snd (write (ds!n) (snd (fold_map write (take n ds) m0))))
        → 1 ∉ | (finsert (fst (write (adata.Array ds) m0)) s)"
        using range_notin_s[OF <n < length ds> <n < length xs> xs3 Array(2)] by blast
      ultimately have "s_disj_fs (loc (snd (fold_map write (take n ds) m0)))
        (arange_safe (finsert (fst (write (adata.Array ds) m0)) s)
        (snd (write (ds!n) (snd (fold_map write (take n ds) m0))))
        (fst (write (ds!n) (snd (fold_map write (take n ds) m0)))))"
        using Array by blast
      then have "s_disj_fs (loc (snd (fold_map write (take n ds) m0)))
        (arange_safe (finsert (fst (write (adata.Array ds) m0)) s)
        (snd (write (ds!n) (snd (fold_map write (take n ds) m0))))
        (xs ! n))"
        using xs3 by (metis Suc.prem1 Suc.le_eq)
      moreover from <n < length ds> have "prefix
        (snd (write (ds ! n) (snd (fold_map write (take n ds) m0))))
        (snd (write (adata.Array ds) m0))"
        using write_sprefix_take sprefix_prefix by blast
      ultimately have "s_disj_fs (loc (snd (fold_map write (take n ds) m0)))
        (arange_safe (finsert (fst (write (adata.Array ds) m0)) s)
        (snd (write (adata.Array ds) m0))
        (xs ! n))"
        using a_data.range_safe_prefix[of "(snd (write (ds ! n) (snd (fold_map write (take n ds)
m0))))"]
        unfolding s_disj_fs_def pred_some_def by (auto simp del: a_data.range_safe.simps)
      moreover have "prefix m0 (snd (fold_map write (take n ds) m0))"
        using fold_map_prefix[of "write"]
        by (metis write_sprefix prod.collapse sndI sprefix_prefix)
      ultimately show ?thesis
        using s_disj_fs_prefix[of m0 " (snd (fold_map write (take n ds) m0))"] by blast
    qed
  ultimately show "s_disj_fs (loc m0) (?f (xs!n) x)"

```

```

      by (auto simp add:s_disj_fs_def pred_some_def)
    qed
  ultimately show ?case by simp
  qed
  qed
  qed
  ultimately show ?case by simp
qed

corollary write_arange:
  assumes "write cd m0 = (l, m)"
  shows "s_disj_fs (loc m0) (arange m l)"
  using write_arange_safe
  unfolding a_data.range_def
  by (metis assms fempty_iff fst_conv snd_conv)

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))))))"
  using assms(3,1,2)
proof -
  from assms write_obtain[of ds m0] obtain l' m'
  where *: "write (ds ! j) (snd (fold_map write (take j ds) m0)) = (l', m)'"
  and **: "prefix m' m"
  by (metis K.cases snd_conv)
  moreover from * have
  ***: "s_disj_fs (loc (snd (fold_map write (take j ds) m0))) (arange m' l)'"
  using write_arange[where ?m0.0="snd (fold_map write (take j ds) m0)"] by auto
  moreover from *** obtain x where "arange m' l' = Some x"
  unfolding s_disj_fs_def pred_some_def by auto
  then have "arange m' l' = arange m l'"
  using assms * ** by (metis a_data.range_def a_data.range_safe_prefix)
  moreover have
  "loc (snd (write (ds ! i) (snd (fold_map write (take i ds) m0))))
  ⊆ loc (snd (fold_map write (take j ds) m0))" using assms
  by (metis (no_types, lifting) Suc_leI fold_map_take_snd loc_write_take order.strict_trans)
  ultimately show ?thesis unfolding s_disj_fs_def pred_some_def by auto
qed

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 ∈ |s| the (arange_safe s (snd (write cd m0)) (fst (write cd m0))).
    x < (length (snd (write cd m0))))"
  using assms
proof (induction cd arbitrary: m0 s)
  case (Value x)
  then show ?case
  by (auto simp add: s_union_fs_def pred_some_def length_append_def case_memory_def loc_def)
next
  case (Array ds)
  define f where "f =
    (λx y. y
      ≧≧ (λy'.
        (arange_safe (finsert (fst (write (Array ds) m0)) s)
          (snd (write (Array ds) m0)) x)

```

```

    >> (λl. Some (l |∪| y'))))"
from write_obtain obtain xs
  where xs1: "snd (write (Array ds) m0) $ fst (write (Array ds) m0) = Some (mdata.Array xs)"
    and xs2: "length xs = length ds"
    and xs3: "∀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 (adata.Array
ds) m0))"
  by metis

have xs0: "¬ fst (write (Array ds) m0) |∈| s" using Array(2)
  by (metis less_Suc_eq_le write_length_inc write_length_suc nth_safe_length xs1)
then have "arange_safe s
  (snd (write (adata.Array ds) m0))
  (fst (write (adata.Array ds) m0))
  = fold f xs (Some {|fst (write (Array ds) m0)|})"
  using xs1 by (simp add:case_memory_def f_def)
moreover have
  "s_union_fs
  (loc (snd (write (adata.Array ds) m0)))
  (loc m0)
  (fold f xs (Some {|fst (write (Array ds) m0)|}))
  ∧ (∀x |∈| the (fold f xs (Some {||})). x < fst (write (adata.Array ds) m0))
  ∧ (fold f xs (Some {||})) ≠ None"
  (is "?UNION ds xs")
proof (rule take_all[where ?P = "λxs' ys'. ?UNION ys' xs'"])
  show "∀n ≤ length xs. ?UNION (take n ds) (take n xs)"
  proof (rule allI, rule impI)
    fix n
    assume "n ≤ length xs"
    then show "?UNION (take n ds) (take n xs)"
    proof (induction n)
      case 0
      then show ?case by (auto simp add:s_union_fs_def pred_some_def loc_def length_append_def)
    next
      case (Suc n)
      then have "n < length xs" by simp
      then have "n < length ds" "n ≤ length xs" using xs2 by simp+
      have *: "prefix (take n ds) (take (Suc n) ds)"
        unfolding prefix_def Suc(1) by (metis <n < length ds> take_hd_drop)

      let ?s="(finsert (fst (write (adata.Array ds) m0)) s)"
      let ?B="(snd (write (ds ! n) (butlast (snd (write (adata.Array (take n ds) m0))))))"
      let ?C="(fst (write (ds ! n) (butlast (snd (write (adata.Array (take n ds) m0))))))"
      let ?A="(butlast (snd (write (adata.Array (take n ds) m0))))"

      have a3: "(fold f (take n xs) (Some {||})) ≠ None" using Suc by simp

      have
        "∀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"
          using length_write_write[OF <n < length ds> Array(2)] xs3
          by (meson <n < length xs>)
      then have a4: "arange_safe ?s ?B ?C ≠ None"
        using Array(1)[of
          "ds!n"
          "(butlast (snd (write (adata.Array (take n ds) m0))))"
          "(finsert (fst (write (adata.Array ds) m0)) s)"]
          unfolding s_union_fs_def pred_some_def
          using <n < length ds> nth_mem by blast
      from a4 have a5: "arange_safe ?s
        (snd (write (Array ds) m0))"

```

```

      ((fst (write (ds!n) (snd (fold_map write (take n ds) m0)))) ≠ None"
using xs3 <n ≤ length xs>
  a_data.range_safe_prefix[of
    "(snd (write (ds ! n) (butlast (snd (write (adata.Array (take n ds)) m0)))))"
    "(snd (write (Array ds) m0))" "(finsert (fst (write (adata.Array ds) m0)) s)"]
  butlast_write[of "(take n ds)" m0]
apply (auto simp del:write.simps a_data.range_safe.simps)
by (metis <n < length xs>)
have a6:
  "fset (the (fold f (take n xs) (Some {||})))
    = loc (butlast (snd (write (adata.Array (take n ds)) m0))) - loc m0"
proof (rule s_disj_union_fs)
show "s_disj_fs (loc m0) (fold f (take n xs) (Some {||})))"
proof -
  have
    "s_disj_fs
      (loc m0)
      (arange_safe s (snd (write (adata.Array ds) m0)) (fst (write (adata.Array ds) m0)))"
  using write_arange_safe[OF Array(2)] by simp
  then have "s_disj_fs (loc m0) (fold f xs (Some {fst (write (Array ds) m0)|}))"
  using xs0 xs1 unfolding f_def by (auto simp add: case_memory_def)
  moreover have "s_disj_fs (loc m0) (Some {fst (write (adata.Array ds) m0)|})"
  unfolding s_disj_fs_def pred_some_def loc_def
  using fold_map_mono write_length_inc[of m0 "(adata.Array ds)"]
  by (auto split:prod.split simp add:length_append_def)
  ultimately have
    "s_disj_fs
      (loc m0)
      (fold f (take n xs) (Some {fst (write (Array ds) m0)|}))"
  unfolding f_def
  using s_disj_fs_loc_fold[of
    m0
    "arange_safe (finsert (fst (write (adata.Array ds) m0)) s) (snd (write (adata.Array
ds) m0))]"
    xs "Some {fst (write (adata.Array ds) m0)|}" n] by simp
  moreover from a3 have "the (fold f (take n xs) (Some {fst (write (Array ds) m0)|})) =
the (fold f (take n xs) (Some {||})) |∪| {fst (write (Array ds) m0)|}"
  unfolding f_def using fold_none_the_fold[of
    "arange_safe (finsert (fst (write (adata.Array ds) m0)) s) (snd (write (adata.Array
ds) m0))]"
    "(take n xs)" "{||}" "{fst (write (adata.Array ds) m0)|}" by auto
  ultimately have "s_disj_fs (loc m0) (fold f (take n xs) (Some {||}))"
  unfolding s_disj_fs_def pred_some_def
  using a3 by auto
  then show ?thesis unfolding f_def
  using s_disj_fs_loc_fold[of
    m0
    "arange_safe (finsert (fst (write (adata.Array ds) m0)) s) (snd (write (adata.Array ds)
m0))]"
    xs "Some {||}" n] by simp
  qed
next
from Suc(1)[OF <n ≤ length xs>] have
  "s_union_fs
    (loc (snd (write (adata.Array (take n ds)) m0)))
    (loc m0)
    (fold f (take n xs) (Some {fst (write (adata.Array (take n ds)) m0)|}))"
  by simp
  then show
    "s_union_fs
      (loc (butlast (snd (write (adata.Array (take n ds)) m0))))
      (loc m0)
      (fold f (take n xs) (Some {||}))"
  proof (rule s_union_fs_s_union_fs_diff[OF _ _ a3])

```

```

show "loc (butlast (snd (write (adata.Array (take n ds)) m0))) =
  loc (snd (write (adata.Array (take n ds)) m0)) - {fst (write (adata.Array (take n ds))
m0)}"

  unfolding loc_def using write_length_suc[of "adata.Array (take n ds)" m0] by fastforce
next
have "fst (write (adata.Array (take n ds)) m0) | $\notin$ |
the (fold f (take n xs) (Some {||}))" using Suc(1)[OF <n  $\leq$  length xs>]
  unfolding loc_def by blast
then show "the (fold f (take n xs) (Some {||})) =
the (fold f (take n xs) (Some {fst (write (adata.Array (take n ds)) m0)|})) | $\neq$ |
{fst (write (adata.Array (take n ds)) m0)|}"
  using fold_insert_same[of "fst (write (adata.Array (take n ds)) m0)" _ n xs "{||}"]
  unfolding f_def by blast
next
show "fst (write (adata.Array (take n ds)) m0)  $\notin$  loc m0"
  by (metis loc_def mem_Collect_eq write_length_inc write_length_suc not_less_eq)
qed
qed

have a7:"
  the ((
    arange_safe ?s
      (snd (write (Array ds) m0))
      ((fst (write (ds!n) (snd (fold_map write (take n ds) m0))))))
  | $\cap$ | the (fold f (take n xs) (Some {||})) = {||}"
proof -
  have
    "arange_safe ?s
      (snd (write (Array ds) m0))
      ((fst (write (ds!n) (snd (fold_map write (take n ds) m0))))))
= arange_safe (finsert (fst (write (adata.Array ds) m0)) s) ?B ?C"
proof -
  have "prefix ?B (snd (write (Array ds) m0))"
    using xs3 by (metis <n < length xs> butlast_write)
  then have
    "arange_safe ?s ?B ?C
    = arange_safe ?s (snd (write (adata.Array ds) m0)) ?C"
    using a_data.range_safe_prefix[of ?B "(snd (write (Array ds) m0))" ?s] a4
    by fastforce
  moreover have
    "butlast (snd (write (adata.Array (take n ds)) m0))
    = snd (fold_map write (take n ds) m0)"
    using butlast_write by auto
  ultimately show ?thesis by auto
qed
moreover have "loc ?A  $\cap$  fset (the (arange_safe ?s ?B ?C)) = {}"
proof -
  have "s_disj_fs (loc ?A) (arange_safe ?s ?B ?C)"
  proof (rule write_arange_safe)
    show " $\forall l \geq \text{length } ?A. l < \text{length } ?B \longrightarrow l \notin ?s$ "
      using length_write_write[OF <n < length ds> Array(2)] xs3 <n < length xs> by blast
  qed
  then show ?thesis unfolding s_disj_fs_def pred_some_def by auto
qed
ultimately show ?thesis using a6 by fastforce
qed

show ?case
proof
  show "s_union_fs (loc (snd (write (adata.Array (take (Suc n) ds)) m0))) (loc m0)
(fold f (take (Suc n) xs) (Some {fst (write (adata.Array (take (Suc n) ds)) m0)|}))"
  proof (rule s_union_fs_s_union_fs_union[OF conjunct1[OF Suc(1)[OF <n  $\leq$  length xs>]]])
    show "(loc (snd (write (adata.Array (take (Suc n) ds)) m0))) =
loc (snd (write (adata.Array (take n ds)) m0))"

```

```

- {length (snd (write (adata.Array (take n ds)) m0)) - 1}
∪ (insert (length (snd (write (ds!n) (butlast (snd (write (adata.Array (take n ds))
m0))))))
    (loc (snd (write (ds!n) (butlast (snd (write (adata.Array (take n ds)) m0))))))
    - (loc (butlast (snd (write (adata.Array (take n ds)) m0))))))"
apply (auto simp add:loc_def length_append_def split:prod.split)
using fold_map_take_snd[OF <n < length ds>] apply (metis less_Suc_eq snd_conv)
using fold_map_take_snd[OF <n < length ds>] apply (metis less_Suc_eq snd_conv)
using fold_map_take_snd[OF <n < length ds>] apply (metis lessI snd_conv)
using write_fold_map_mono[OF *, of m0] unfolding prefix_def apply auto[1]
using fold_map_take_snd[OF <n < length ds>] apply (metis less_SucI snd_conv)
done
next
show "{length (snd (write (adata.Array (take n ds)) m0)) - 1} ∩ loc m0 = {}"
  using write_length_inc[of m0 "(adata.Array (take n ds))"] unfolding loc_def by simp
next
show "{length (snd (write (adata.Array (take n ds)) m0)) - 1}
= fset {(fst (write (Array (take n ds)) m0))}"
  using write_length_suc[of "adata.Array (take n ds)"] by simp
next
show "insert (length ?B) (loc ?B - loc ?A)
= fset (the (Some (finsert
  (fst (write (adata.Array (take (Suc n) ds)) m0))
  (the (f (xs!n) (fold f (take n xs) (Some {||}))))
  |-| (the (fold f (take n xs) (Some {||})))))))"
proof -
have "length ?B = (fst (write (adata.Array (take (Suc n) ds)) m0))"
  using write_Array_take_Suc
  by (metis <n < length ds>)
moreover have "(loc ?B - loc ?A)
= fset (the (f (xs!n) (fold f (take n xs) (Some {||}))))
|-| (the (fold f (take n xs) (Some {||}))))"
proof -
have "(loc ?B - loc ?A)
= fset (the (arange_safe (finsert (fst (write (adata.Array ds) m0)) s) ?B ?C))"
proof (rule s_union_fs_diff)
have "ds ! n ∈ set ds"
  by (simp add: <n < length ds>)
moreover have "∀l ≥ length ?A. l < length ?B → l ∉ ?s"
  using length_write_write[OF <n < length ds> Array(2)] <n < length xs> xs3
  by blast
ultimately show "s_union_fs (loc ?B) (loc ?A) (arange_safe ?s ?B ?C)"
  using Array(1) by blast
next
show "loc ?A ∩ fset (the (arange_safe ?s ?B ?C)) = {}"
proof -
have "s_disj_fs (loc ?A) (arange_safe ?s ?B ?C)"
proof (rule write_arange_safe)
  show "∀l ≥ length ?A. l < length ?B → l ∉ ?s"
  using length_write_write[OF <n < length ds> Array(2)] xs3 <n < length xs>
  by blast
qed
then show ?thesis unfolding s_disj_fs_def pred_some_def by auto
qed
qed
moreover have
"fset (the (arange_safe ?s ?B ?C))
= fset (the (f (xs!n) (fold f (take n xs) (Some {||}))))
|-| (the (fold f (take n xs) (Some {||}))))"
proof -
from a3 have
"(λx y. y ≫= (λy'. (arange_safe ?s (snd (write (Array ds) m0)) x)
  ≫= (λl. Some (l |∪| y')))) (xs!n) (fold f (take n xs) (Some {||}))
= (arange_safe ?s

```

```

      (snd (write (Array ds) m0))
      ((fst (write (ds!n) (snd (fold_map write (take n ds) m0))))))
    >>= (λl. Some (1 |∪| the (fold f (take n xs) (Some {||}))))"
  using xs3 <n < length xs> by fastforce
  moreover from a5 have
    "(arange_safe ?s
      (snd (write (Array ds) m0))
      ((fst (write (ds!n) (snd (fold_map write (take n ds) m0))))))
    >>= (λl. Some (1 |∪| the (fold f (take n xs) (Some {||}))))
    = Some (the (
      arange_safe ?s
      (snd (write (Array ds) m0))
      ((fst (write (ds!n) (snd (fold_map write (take n ds) m0))))))
      |∪| the (fold f (take n xs) (Some {||}))))"
  by fastforce
  moreover from a7 have
    "the
      (arange_safe ?s
      (snd (write (Array ds) m0))
      ((fst (write (ds!n) (snd (fold_map write (take n ds) m0))))))
      |∪| the (fold f (take n xs) (Some {||}))
      |-| (the (fold f (take n xs) (Some {||})))
    = the
      (arange_safe ?s
      (snd (write (Array ds) m0))
      ((fst (write (ds!n) (snd (fold_map write (take n ds) m0))))))"
  by blast
  ultimately have
    "the (
      (λx y. y >>= (λy'. (arange_safe ?s (snd (write (Array ds) m0)) x)
        >>= (λl. Some (1 |∪| y')))) (xs!n) (fold f (take n xs) (Some {||}))
      |-| (the (fold f (take n xs) (Some {||})))
    = the (
      arange_safe ?s
      (snd (write (Array ds) m0))
      (fst (write (ds!n) (snd (fold_map write (take n ds) m0))))))"
  by simp
  moreover have "prefix ?B (snd (write (Array ds) m0))"
  using xs3 by (metis <n < length xs> butlast_write)
  moreover have "?C = (fst (write (ds!n) (snd (fold_map write (take n ds) m0))))"
  by (metis butlast_write)
  ultimately show ?thesis
  using a_data.range_safe_prefix[of ?B "(snd (write (Array ds) m0))" ?s]
  f_def a4 by force
  qed
  ultimately show ?thesis by blast
  qed
  ultimately show ?thesis by simp
  qed
next
show "fold f (take (Suc n) xs) (Some {fst (write (adata.Array (take (Suc n) ds)) m0)|}) =
  Some (the (fold f (take n xs) (Some {fst (write (adata.Array (take n ds)) m0)|}))
  |-| {fst (write (adata.Array (take n ds)) m0)|})
  |∪| the (Some (finsert
    (fst (write (adata.Array (take (Suc n) ds)) m0))
    (the (f (xs ! n) (fold f (take n xs) (Some {||})))
    |-| the (fold f (take n xs) (Some {||}))))))"
(is "?f (Suc n) = Some (the (?f n) |-| {!s n|} |∪| the (Some (finsert (?s (Suc n)) ?r))))"
proof -
  have "?f (Suc n) = fold f (take (Suc n) xs) (Some {||}) >>= Some o finsert (?s (Suc n))"
  using fold_some_some Suc(2) unfolding f_def by fast
  also have
    "fold f (take (Suc n) xs) (Some {||})
    = f (xs ! n) (fold f (take n xs) (Some {||}))"

```

```

    using fold_take[OF <n < length xs>] .
  also have "(fold f (take n xs) (Some {||})) = Some ((the (?f n) |-| {!s n!}))"
    unfolding f_def
  proof (rule fold_some_diff)
    have "fst (write (adata.Array (take n ds)) m0)
      ≠ loc (butlast (snd (write (adata.Array (take n ds)) m0)))"
      unfolding loc_def using write_length_suc[of "(adata.Array (take n ds))"] by simp
    then show
      "fst (write (adata.Array (take n ds)) m0) |∈|
      the (fold
        (λx y. y ≧=
          (λy'.
            arrange_safe
              (finsert (fst (write (adata.Array ds) m0)) s)
                (snd (write (adata.Array ds) m0))
                  x
                ≧= (λl. Some (l |∪| y')))))
          (take n xs) (Some {||}))"
      using a6 unfolding f_def by simp
    next
      from a3 show
        "fold (λx y. y ≧=
          (λy'. arrange_safe (finsert (fst (write (adata.Array ds) m0)) s)
            (snd (write (adata.Array ds) m0)) x ≧= (λl. Some (l |∪| y')))) (take n xs) (Some
{||}))
          ≠ None"
          unfolding f_def by auto
    qed
  finally have
    "?f (Suc n) = f (xs ! n) (Some (the (?f n) |-| {!s n!})) ≧= Some o finsert (?s (Suc
n))" .
  moreover from a7 have
    "the (arrange_safe (finsert (fst (write (adata.Array ds) m0)) s)
      (snd (write (adata.Array ds) m0)) (xs ! n))
    |∩| the (fold f (take n xs) (Some {||})) = {||}" using xs3 f_def by (metis <n < length
xs>)
  moreover from a5 have
    "arrange_safe (finsert (fst (write (adata.Array ds) m0)) s)
      (snd (write (adata.Array ds) m0)) (xs ! n) ≠ None" using xs3
    by (simp add: <n < length xs>)
  ultimately show ?thesis using a3 unfolding f_def by auto
  qed
  qed
  next
  show
    "(∀x|∈|the (fold f (take (Suc n) xs) (Some {||})).
      x < fst (write (adata.Array (take (Suc n) ds)) m0))
    ∧ fold f (take (Suc n) xs) (Some {||}) ≠ None"
  proof
    show
      "∀x|∈|the (fold f (take (Suc n) xs) (Some {||})).
        x < fst (write (adata.Array (take (Suc n) ds)) m0)"
    proof (rule ballI)
      fix x assume "x |∈| the (fold f (take (Suc n) xs) (Some {||}))"
      then have
        "x |∈| the ((λx y. y
          ≧= (λy'.
            arrange_safe (finsert (fst (write (Array ds) m0)) s)
              (snd (write (Array ds) m0)) x
            ≧= (λl. Some (l |∪| y')))) (xs ! n) (fold f (take n xs) (Some {||})))"
        using fold_take[OF <n < length xs>]
        unfolding f_def by (rule back_subst)
      moreover obtain y where y_def: "fold f (take n xs) (Some {||}) = Some y"
        using a3 by auto
    qed
  qed

```

```

moreover obtain z
  where z_def:
    "arange_safe
      (finsert (fst (write (adata.Array ds) m0)) s)
      (snd (write (adata.Array ds) m0))
      (xs ! n)
    = Some z"
  using a5 xs3 using <n < length xs> by auto
ultimately consider "x |∈| y" | "x |∈| z"
  by (auto simp del:a_data.range_safe.simps write.simps)
then show "x < fst (write (adata.Array (take (Suc n) ds)) m0)"
proof cases
  case 1
  moreover have
    "fst (write (adata.Array (take n ds)) m0)
      < fst (write (adata.Array (take (Suc n) ds)) m0)"
  apply (auto split:prod.split simp add:length_append_def)
  by (metis <n < length ds> fold_map_take_snd write_length_inc snd_eqD)
  ultimately show ?thesis using Suc(1)[OF <n ≤ length xs>] y_def by auto
next
  case 2
  have
    "∀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"
  using range_notin_s[OF <n < length ds> <n < length xs> _ Array(2)] xs3 by blast
  then have "(∀x|∈|the (arange_safe (finsert (fst (write (adata.Array ds) m0)) s)
    (snd (write (ds ! n) (snd (fold_map write (take n ds) m0))))
    (fst (write (ds ! n) (snd (fold_map write (take n ds) m0)))))).
    x < (length (snd (write (ds ! n) (snd (fold_map write (take n ds) m0))))))"
  using
    Array(1)[of
      "(ds ! n)"
      "(snd (fold_map write (take n ds) m0))"
      "(finsert (fst (write (adata.Array ds) m0)) s)"]
    <n < length ds> nth_mem by blast
  moreover from a4 have
    "x|∈|the (arange_safe (finsert (fst (write (adata.Array ds) m0)) s)
      (snd (write (ds ! n) (snd (fold_map write (take n ds) m0))))
      (fst (write (ds ! n) (snd (fold_map write (take n ds) m0))))))"
  using 2 xs3 z_def <n < length xs> butlast_write a_data.range_safe_prefix
  by (smt (verit, del_insts) option.exhaust_sel option.sel)
  ultimately have
    "x < (length (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))))"
  by simp
  then show ?thesis
  apply (auto split:prod.split simp add:length_append_def)
  using fold_map_take_snd[OF <n < length ds>, of "write" m0] by simp
  qed
qed
next
  have "f (xs ! n) (fold f (take n xs) (Some {||})) ≠ None"
  proof -
    from Suc obtain x where "(fold f (take n xs) (Some {||})) = x" by simp
    then show ?thesis using <n < length xs> a3 a5 not_None_eq xs3 unfolding f_def by auto
  qed
  then show "fold f (take (Suc n) xs) (Some {||}) ≠ None"
  using fold_take[OF <n < length xs>, of f "(Some {||})"]
  by simp
  qed
  qed
  qed
  qed
  qed (rule xs2)

```

```

moreover have
  "( $\forall x \in \text{the (fold f xs (Some \{fst (write (adata.Array ds) m0\}))}$ ).
    x < (length (snd (write (adata.Array ds) m0))))"
proof
  fix x
  assume *: "x  $\in$  the (fold f xs (Some \{fst (write (adata.Array ds) m0\})))"
  have "fold f xs (Some (\{\}  $\cup$  \{fst (write (adata.Array ds) m0\}))  $\neq$  None"
    using calculation(2) unfolding s_union_fs_def pred_some_def by auto
  with *
  consider "x  $\in$  the (fold f xs (Some \{\}))"
    | "x  $\in$  \{fst (write (adata.Array ds) m0)\}"
    using fold_none_the_fold[where ?X="\{\}" and ?Y="\{fst (write (adata.Array ds) m0)\}"]
    unfolding f_def by fastforce
  then show "x < (length (snd (write (adata.Array ds) m0)))"
  proof cases
    case 1
    then show ?thesis using calculation(2) write_length_suc[of "(adata.Array ds)" m0] by auto
  next
    case 2
    then show ?thesis
      by (metis fsingletonE lessI write_length_suc)
  qed
qed
ultimately show ?case by simp
qed

corollary write_loc:
  assumes "write cd m0 = (l, m)"
  shows "s_union_fs (loc m) (loc m0) (arange m l)"
  using assms write_loc_safe unfolding a_data.range_safe_prefix
  by (metis a_data.range_def fempty_iff fst_conv snd_conv)

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_subst_s
    (arange m i')
    (loc (snd (write (ds ! i) (snd (fold_map write (take i ds) m0)))))"
  using assms
proof -
  from assms(3)
  have
    "s_union_fs
      (loc (snd (write (ds ! i) (snd (fold_map write (take i ds) m0))))
      (loc (snd (fold_map write (take i ds) m0)))
      (arange (snd (write (ds ! i) (snd (fold_map write (take i ds) m0)))) i'))"
  using write_loc[of
    "ds ! i"
    "snd (fold_map write (take i ds) m0)"
    i'
    "snd (write (ds ! i) (snd (fold_map write (take i ds) m0)))]"
  by simp
  then obtain L
  where L_def: "arange (snd (write (ds ! i) (snd (fold_map write (take i ds) m0)))) i'
    = Some L"
    and "loc (snd (write (ds ! i) (snd (fold_map write (take i ds) m0))))
    = loc (snd (fold_map write (take i ds) m0))  $\cup$  fset L"
  unfolding s_union_fs_def pred_some_def by blast
  moreover from assms(1,2) write_obtain[of ds m0] have
    "prefix (snd (write (ds ! i) (snd (fold_map write (take i ds) m0)))) m" by auto
  with L_def have "arange m i' = Some L" using a_data.range_prefix by blast
  ultimately show ?thesis unfolding fs_subst_s_def pred_some_def by blast
qed

```

```

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 -
  from assms(3) obtain LL
  where "arange_safe s (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) y = Some LL"
  using write_loc_safe[OF assms(4)] unfolding s_union_fs_def pred_some_def by blast
  then show ?thesis using assms(1,2)
  by (metis a_data.range_safe_prefix)
qed

```

```

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 -
  from assms(2) obtain L
  where "arange (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) y = Some L"
    and subs: "fset L ⊆ loc (snd (write (ds ! n) (snd (fold_map write (take n ds) m0))))"
  using write_loc[of "ds ! n" "snd (fold_map write (take n ds) m0)"]
  unfolding s_union_fs_def pred_some_def by force
  then have "arange m y = Some L" using a_data.range_prefix[OF assms(5)] by auto
  with assms(4) have "x |∈| L"
  by (metis assms(3) bot.extremum a_data.range_def option.inject a_data.range_safe_subset_same)
  with subs obtain l
  where "snd (write (ds ! n) (snd (fold_map write (take n ds) m0))) $ x = Some l"
  unfolding loc_def by auto
  with assms(1,5) show ?thesis by (metis nth_safe_prefix)
qed

```

3.20 Memory Init and Memory Copy (Memory)

```

theorem write_ared_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"
  using assms
proof (induction cd arbitrary: m0 s)
  case (Value x)
  then show ?case by (auto simp add: length_append_def case_memory_def prefix_def)
next
  case (Array ds)
  show ?case
  proof (rule allI, rule impI)
    fix mx
    assume 1: "prefix (snd (write (adata.Array ds) m0)) mx"
    from write_obtain obtain xs
    where xs1: "snd (write (Array ds) m0) $ fst (write (Array ds) m0) = Some (mdata.Array xs)"
      and xs2: "length xs = length ds"
      and xs3: "\n < length xs. xs!n < fst (write (Array ds) m0)
        ∧ xs!n = fst (write (ds!n) (snd (fold_map write (take n ds) m0)))"
    by metis
    moreover have "\fst (write (Array ds) m0) |∈| s" using Array(2)
    by (metis less_Suc_eq_le write_length_inc write_length_suc nth_safe_length xs1)
    ultimately have "ared_safe s

```

```

      (snd (write (adata.Array ds) m0))
      (fst (write (adata.Array ds) m0))
    = those (map (aread_safe
      (finsert (fst (write (Array ds) m0)) s)
      (snd (write (Array ds) m0)))) xs)
    ≧≧ Some ◦ Array" by (simp add:case_memory_def)
moreover have "those (map (aread_safe
  (finsert (fst (write (Array ds) m0)) s)
  (snd (write (Array ds) m0)))) xs)
  = Some ds"
proof (rule take_all[where ?P = "λxs ys. those (map (aread_safe
  (finsert (fst (write (Array ds) m0)) s)
  (snd (write (Array ds) m0)))) xs) = Some ys"]])
show "∀n ≤ length xs. those (map (aread_safe
  (finsert (fst (write (Array ds) m0)) s)
  (snd (write (Array ds) m0)))) (take n xs))
  = Some (take n ds)"
proof (rule allI, rule impI)
  fix n
  assume "n ≤ length xs"
  then show "those (map (aread_safe
    (finsert (fst (write (Array ds) m0)) s)
    (snd (write (Array ds) m0)))) (take n xs))
    = Some (take n ds)"
  proof (induction n)
    case 0
    then show ?case by simp
  next
    case (Suc n)
    from Suc(2) have "n < length xs" by auto
    then have *: "take (Suc n) xs = (take n xs) @ [xs!n]"
      and **: "take (Suc n) ds = (take n ds) @ [ds!n]"
    apply (rule List.take_Suc_conv_app_nth)
    using <n < length xs> take_Suc_conv_app_nth xs2 by auto
    moreover have ***:
      "aread_safe
        (finsert (fst (write (Array ds) m0)) s)
        (snd (write (Array ds) m0))
        (xs!n)
      = Some (ds!n)"
    proof -
      have "aread_safe (finsert (fst (write (Array ds) m0)) s)
        (snd (write (Array ds) m0))
        (fst (write (ds!n) (snd (fold_map write (take n ds) m0))))
      = Some (ds!n)"
    proof -
      have "ds!n ∈ set ds" using <n < length xs> xs2 by auto
      moreover from <n < length xs> have "n < length ds" using xs2 by simp
      then have
        "prefix
          (snd (write (ds!n) (snd (fold_map write (take n ds) m0))))
          (snd (write (Array ds) m0))"
        using write_sprefix_take using sprefix_prefix by blast
      moreover have "∀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 (Array ds) m0)) s)"
      using range_notin_s[OF <n < length ds> <n < length xs> xs3 Array(2)] by blast
      ultimately show ?thesis using Array(1) by blast
    qed
    moreover have "xs!n = fst (write (ds!n) (snd (fold_map write (take n ds) m0)))"
      using xs3 <n < length xs> by simp
    ultimately show ?thesis by simp
  qed
  moreover from <n < length xs> have "n ≤ length xs" by simp

```

```

then have
  "those (map (aread_safe (finsert (fst (write (adata.Array ds) m0)) s)
    (snd (write (adata.Array ds) m0)))
    (take n xs))
  = Some (take n ds)" using Suc(1) xs2 by argo
ultimately show
  "those (map (aread_safe (finsert (fst (write (adata.Array ds) m0)) s)
    (snd (write (adata.Array ds) m0)))
    (take (Suc n) xs))
  = Some (take (Suc n) ds)" using those_those <n < length xs>
  by fastforce
qed
qed
qed (rule xs2)
ultimately show "aread_safe s mx (fst (write (adata.Array ds) m0)) = Some (adata.Array ds)"
  by (metis 1 bind.bind_lunit comp_apply a_data.read_safe_prefix)
qed
qed

corollary write_read:
  assumes "write cd m0 = (l, m)"
    and "prefix m mx"
  shows "aread mx l = Some cd"
  using assms write_aread_safe unfolding a_data.read_def
  by (metis empty_iff fst_conv snd_conv)

```

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 " $\forall l \geq \text{length } m0. l < \text{length } (\text{snd } (\text{write } cd \ m0)) \longrightarrow \neg l \in |s|$ "
  shows "adisjoint m1 L"
  using assms
proof (induction arbitrary: L l m0 rule:write.induct)
  case (1 x m)
  then show ?case unfolding a_data.disjoint_def
    by (auto simp add:length_append_def case_memory_def split:if_split_asm)
next
  case (2 ds m)
  have " $\forall x \in |L|. \forall xs. m \$ x = \text{Some } (mdata.Array \ xs)$ "
     $\longrightarrow (\forall i \ j \ i' \ j' \ L \ L'.
      i \neq j \wedge xs \ \$ \ i = \text{Some } i'
      \wedge xs \ \$ \ j = \text{Some } j'
      \wedge \text{arange } m \ i' = \text{Some } L
      \wedge \text{arange } m \ j' = \text{Some } L'
      \longrightarrow L \ | \cap \ L' = \{\} )$ "
  proof
    fix x
    assume "x  $\in$  |L|"
    then consider
      (1) "x = l"
      | (2) n y L'
    where "n < length ds"
      and "fst (write (ds ! n) (snd (fold_map write (take n ds) m0))) = y"
      and "arange_safe s m y = Some L'"
      and "x  $\in$  |L'"
    using write_range_safe_in "2.premis"(1,2) by blast
  then show
    " $\forall xs. m \$ x = \text{Some } (mdata.Array \ xs)$ "
     $\longrightarrow (\forall i \ j \ i' \ j' \ L \ L'.
      i \neq j
      \wedge xs \ \$ \ i = \text{Some } i'
      \wedge xs \ \$ \ j = \text{Some } j')$ 
  end

```

```

    ∧ arange m i' = Some L
    ∧ arange m j' = Some L'
    → L |∩| L' = {|}|)"

```

proof cases

case 1

show ?thesis

proof (rule, rule, rule, rule, rule, rule, rule, rule, rule, rule)

fix xs i j i' j' L L'

assume *: "m \$ x = Some (mdata.Array xs)"

and **: "i ≠ j"

∧ xs \$ i = Some i'

∧ xs \$ j = Some j'

∧ arange m i' = Some L

∧ arange m j' = Some L'"

{

fix i::nat and j and i' and j' and L and L'

assume "i < j"

and **: "xs \$ i = Some i'"

∧ xs \$ j = Some j'

∧ arange m i' = Some L

∧ arange m j' = Some L'"

moreover from 2(2) **have**

"snd (write (adata.Array ds) m0) \$ fst (write (adata.Array ds) m0)
= Some (mdata.Array xs)"

using * 1 **by** simp

then have

"length xs = length ds"

and 0: "∀n<length xs.

xs ! n < fst (write (adata.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 (adata.Array ds) m0))"

using write_obtain[of ds m0] *

by (fastforce, metis (mono_tags, lifting) mdata.inject(2) option.inject)

then have

"j' = fst (write (ds ! j) (snd (fold_map write (take j ds) m0)))"

and **: "prefix

(snd (write (ds ! j) (snd (fold_map write (take j ds) m0))))

(snd (write (adata.Array ds) m0))"

by (metis "***" nth_safe_def nth_safe_length option.sel)+

moreover have "j < length ds" **using** 'length xs = length ds'

by (metis "***" nth_safe_length)

then have

"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))))))"

using fold_map_write_arange[OF 2(2), of j i] 'i < j' **by** blast

ultimately have

"s_disj_fs

(loc (snd (write (ds ! i) (snd (fold_map write (take i ds) m0))))

(Some L'))"

using ** **by** simp

moreover have

"fset L ⊆ (loc (snd (write (ds ! i) (snd (fold_map write (take i ds) m0)))))"

proof -

from 0 **have** "i' = fst (write (ds ! i) (snd (fold_map write (take i ds) m0)))"

by (metis "***" nth_safe_def nth_safe_length option.sel)+

moreover from 'j < length ds' **have** "i < length ds" **using** 'i < j' **by** simp

ultimately have

"fs_subst_s

(arange m i')

(loc (snd (write (ds ! i) (snd (fold_map write (take i ds) m0))))))"

using fold_map_write_loc[OF 2(2), of i i'] 2(2) ** 'j < length ds' 'i < j' **by** simp

```

    then show ?thesis unfolding fs_subst_s_def pred_some_def using ** by simp
  qed
  ultimately have "L |∩| L' = {}" unfolding s_disj_fs_def pred_some_def by auto
} note inner=this

from ** consider "i < j" | "j < i" by linarith
then show "L |∩| L' = {}"
proof (cases)
  case _: 1
  then show ?thesis using inner ** by blast
next
  case 2
  then show ?thesis using inner ** by blast
qed
qed
next
case 22: 2
from 22(1) have "ds ! n ∈ set ds" by simp
moreover from 22(2) have
  "write (ds ! n) (snd (fold_map write (take n ds) m0))
  = (y, snd (write (ds ! n) (snd (fold_map write (take n ds) m0))))" by auto
moreover have
  *: "∀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"
  by (metis "2.prem3"(3) "22"(1) butlast_write finsetCI fold_map_length fold_map_take_fst
    length_write_write write_fold_map_less)
moreover have **: "prefix (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) m"
  by (metis "2.prem3"(1) "22"(1) write_obtain_split_pairs)
then have 7:
  "arange_safe s (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) y = Some L'"
  using prefix_write_range_safe_same[OF _ 22(3) _ *] 22(2) by blast
ultimately have
  "adisjoint (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) L'"
  using 2(1)[of
    "ds ! n"
    "(snd (fold_map write (take n ds) m0))"
    y
    "snd (write (ds ! n) (snd (fold_map write (take n ds) m0))]"
    L']
  by simp
moreover from 22 have "x ∈ L'" by blast
ultimately have
  *: "∀xs. (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) $ x
    = Some (mdata.Array xs)
    → (∀i j i' j' L L'.
      i ≠ j
      ∧ xs $ i = Some i'
      ∧ xs $ j = Some j'
      ∧ arange (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) i'
        = Some L
      ∧ arange (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) j'
        = Some L'
      → L |∩| L' = {})"
  by (simp add: a_data.disjoint_def)
show ?thesis
proof (rule, rule, rule, rule, rule, rule, rule, rule)
  fix xs i j i' j' L' L''
  assume ***: "m $ x = Some (mdata.Array xs)"
  and ****: "i ≠ j
    ∧ xs $ i = Some i'
    ∧ xs $ j = Some j'
    ∧ arange m i' = Some L'
    ∧ arange m j' = Some L''"
```

```

****: "(snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) $ x
= Some (mdata.Array xs)"
using prefix_write_nth_same[OF *** _ 22(3,4) **] "22"(2) by auto
moreover have
  "arange (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) i' = Some L'"
using a_data.range_safe_prefix_in_range[OF 7 22(4) ****] **** ** by blast
moreover have
  "arange (snd (write (ds ! n) (snd (fold_map write (take n ds) m0)))) j' = Some L'"
using a_data.range_safe_prefix_in_range[OF 7 22(4) ****] **** ** by blast
ultimately show "L' | $\cap$ | L'" = {||}" using * by blast
qed
qed
qed
then show ?case unfolding a_data.disjoint_def
  by (simp add: a_data.range_def arange_safe_def)
qed

corollary write_adisjoint:
  assumes "write a m = (l, m'"
    and "arange m' l = Some L"
  shows "adisjoint m' L"
using write_adisjoint_safe[OF assms(1), of "{||}"] assms arange_def arange_safe_def data.range_def
by (metis fempty_iff)

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"
by (induction a, auto simp add: map_idI)

lemma eq_iff_stoc:
  "a = b  $\longleftrightarrow$  s_to_c a = s_to_c b"
proof (induction a arbitrary: b)
  case (Value x)
  then show ?case
    by (metis c_to_s.simps(1,2) s_to_c.elims adata.disc(1,2))
next
  case (Array xs)
  show ?case
    apply simp using Array
    by (metis call_data.disc(3,4) call_data.sel(2) list.inj_map_strong
        s_to_c.elims s_to_c.simps(2))
qed

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"
  by pat_completeness auto
termination T
  apply (relation "measure (λ(a,b). size a)")
  apply simp
  apply auto
  by (meson less_irrefl_nat not_less_eq nth_mem size_list_estimation)

lemma T_eq_stoc:
  "T x y = (s_to_c x = y)"
proof (induction x arbitrary: y)
  case (Value x)
  then show ?case
    using T.elims(1) by auto
next
  case (Array xs)
  then show ?case
  proof (cases y)
    case (Value x1)
    then show ?thesis
      by (simp add: Array)
  next
    case _: (Array ys)
    then show ?thesis using Array set_listall[where ?f=s_to_c and ?R = T, of xs ys]
      by simp
  qed
qed

lemma q: "Quotient (=) s_to_c c_to_s T"
proof (rule QuotientI)
  show "∧a. s_to_c (c_to_s a) = a" using stoc_ctos by blast
next
  show "∧a. c_to_s a = c_to_s a" by simp
next
  show "∧r s. (r = s) = (r = r ∧ s = s ∧ s_to_c r = s_to_c s)" using eq_iff_stoc by blast
next
  show "T = (λx y. x = x ∧ s_to_c x = y)" using T_eq_stoc by auto
qed

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 .
lift_definition clookup :: "'v::vtype list ⇒ 'v call_data ⇒ 'v call_data option" is Memory.alookup .

context includes lifting_syntax begin

lemma Value_transfer[transfer_rule]: "(R ==> (pcr_call_data R)) adata.Value call_data.Value"
  apply (auto simp: rel_fun_def pcr_call_data_def relcompp_apply)
  by (meson T.simps(1) adata.rel_inject(1))

lemma Array_transfer[transfer_rule]: "((list_all2 (pcr_call_data R)) ==> (pcr_call_data R))
  adata.Array call_data.Array"
  apply (auto simp: rel_fun_def pcr_call_data_def relcompp_apply)
proof -
  fix x y
  assume "list_all2 (rel_adata R 00 T) x y"
  then show "∃b. rel_adata R (adata.Array x) b ∧ T b (call_data.Array y)"
  proof (induction rule: list_all2_induct)
    case Nil
    then show ?case
      by (metis T_eq_stoc list_ctr_transfer(1) list_map_disc_iff s_to_c.simps(2) adata.rel_inject(2))
  qed

```

```

next
  case (Cons x xs y ys)
  then obtain b where *: "rel_adata R (adata.Array xs) b  $\wedge$  T b (call_data.Array ys)" by blast
  then obtain ys' where ys_def: "b = adata.Array ys'"
    by (metis T.simps(4) adata.exhaust_sel)

  from ys_def have "rel_adata R (adata.Array (x # xs)) (adata.Array (c_to_s y # ys'))"
    by (metis Cons.hyps(1) T_eq_stoc * stoc_ctos eq_iff_stoc list.rel_intros(2) relcompp.cases
adata.rel_inject(2))
  moreover from ys_def have "T (adata.Array (c_to_s y # ys')) (call_data.Array (y # ys))"
    using T_eq_stoc * stoc_ctos by auto
  ultimately show ?case by blast
qed
qed

lemma fold_map_transfer[transfer_rule]: "(A  $\implies$  B  $\implies$  rel_prod C B)  $\implies$  list_all2 A  $\implies$  B  $\implies$ 
rel_prod (list_all2 C) B fold_map fold_map"
  apply (auto simp: rel_fun_def pcr_call_data_def relcompp_apply)
proof -
  fix x y xa ya xb yb
  assume "list_all2 A xa ya"
and "\xa ya. A xa ya  $\longrightarrow$  ( $\forall$ xb yb. B xb yb  $\longrightarrow$  rel_prod C B (x xa xb) (y ya yb))"
  and "B xb yb"
  then show "rel_prod (list_all2 C) B (fold_map x xa xb) (fold_map y ya yb)"
  proof (induction arbitrary: xb yb rule: list_all2_induct)
    case Nil
    then show ?case
      by simp
  next
    case (Cons x' xs y' ys)
    obtain x0 y0 xs0 y0' where 1: "x x' xb = (x0, y0)" and 2: "(fold_map x xs y0) = (xs0, y0')" and
3: "(fold_map x (x' # xs) xb) = (x0 # xs0, y0')" by fastforce
    obtain x1 y1 xs1 y1' where 4: "y y' yb = (x1, y1)" and 5: "(fold_map y ys y1) = (xs1, y1')" and
6: "(fold_map y (y' # ys) yb) = (x1 # xs1, y1')" by fastforce

    from Cons have "rel_prod (list_all2 C) B (fold_map x xs y0) (fold_map y ys y1)"
      using <x x' xb = (x0, y0)> <y y' yb = (x1, y1)> by force
    then have "(list_all2 C) xs0 xs1"
      and "B y0' y1'" using 2 5 by simp+
    moreover have "C x0 x1"
      using Cons.hyps(1) Cons.prems(1,2) 1 4 by fastforce
    then have "(list_all2 C) (x0 # xs0) (x1 # xs1)"
      by (simp add: <list_all2 C xs0 xs1>)
    ultimately show ?case using 3 6 by simp
  qed
qed

lemma eq_transfer[transfer_rule]: "(rel_option (pcr_call_data (=))  $\implies$  (rel_option (pcr_call_data
(=))  $\implies$  (=))) (=) (=)"
  apply (auto simp: rel_fun_def pcr_call_data_def relcompp_apply)
  apply (metis T_eq_stoc eq_00 option.rel_cases option.sel rel_option_None1 adata.rel_eq)
  by (metis T_eq_stoc eq_iff_stoc eq_00 option.rel_eq option.rel_sel adata.rel_eq)

lemma nth_safe_transfer[transfer_rule]: "(list_all2 ((pcr_call_data (=)))  $\implies$  ((=))  $\implies$  rel_option
(pcr_call_data (=)) nth_safe nth_safe"
  apply (auto simp: rel_fun_def pcr_call_data_def relcompp_apply)
  apply (simp add: eq_00 list_all2_conv_all_nth adata.rel_eq)
  by (simp add: nth_safe_def)

end

lemma write_length_append[simp,code]: "write (call_data.Value x) m = length_append m (mdata.Value x)"
  apply transfer
  by simp

```

```

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)))"
  apply transfer
  by auto

lemma clookup[simp,code]:
  "clookup [] s = Some s"
  apply (transfer)
  by simp

lemma clookup2[simp,code]:
  "clookup (i # is) (call_data.Array xs) = vtype_class.to_nat i >>= ($) xs >>= clookup is"
  apply transfer
  by simp

lemma clookup_none[simp,code]:
  "clookup (v # va) (call_data.Value vb) = None"
  apply transfer by simp

lemma write_length_inc: "length (snd (write cd m0)) > length m0"
  apply transfer using write_length_inc by blast

corollary write_loc:
  assumes "write cd m0 = (l, m)"
  shows "s_union_fs (loc m) (loc m0) (arange m l)"
  using assms apply transfer using write_loc by blast

```

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 o 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 o 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
.
```

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  $\Rightarrow$  nat option" where
  "to_nat_valtype (Uint x) = Some (unat x)"
| "to_nat_valtype _ = None"

```

```

instance ..

```

```

end

```

3.27 Common functions (State)

```

fun lift_bool_unary::"(bool  $\Rightarrow$  bool)  $\Rightarrow$  ('a::address) valtype  $\Rightarrow$  ('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  $\Rightarrow$  bool  $\Rightarrow$  bool)  $\Rightarrow$  ('a::address) valtype  $\Rightarrow$  ('a::address) valtype  $\Rightarrow$ 
('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 ( $\wedge$ )"

```

```

definition vtor where
  "vtor = lift_bool_binary ( $\vee$ )"

```

```

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  $\Rightarrow$  256 word  $\Rightarrow$  bool)  $\Rightarrow$  ('a::address) valtype  $\Rightarrow$  ('a::address) valtype  $\Rightarrow$ 
('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  $\Rightarrow$  256 word  $\Rightarrow$  256 word)  $\Rightarrow$  ('a::address) valtype  $\Rightarrow$  ('a::address)
valtype  $\Rightarrow$  ('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  $\Rightarrow$  ('a::address) valtype  $\Rightarrow$  ('a::address) valtype option"
where
  "vtplus_safe (Uint l) (Uint r) = (if unat l + unat r < 2256 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  $\Rightarrow$  ('a::address) valtype  $\Rightarrow$  ('a::address) valtype option"
where
  "vtminus_safe (Uint l) (Uint r) = (if r  $\leq$  l then Some (Uint (l - r)) else None)"

```

3 Stores and State

```
| "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"
  unfolding balances_update_def by simp

```

```

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 o Bool"
```

```
definition kdSint where
  "kdSint ≡ Value o Uint"
```

```
definition kdAddress where
  "kdAddress = Value o Address"
```

```
definition kdBytes where
  "kdBytes ≡ Value o 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 o 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 o 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 kminus where
  "kminus = lift_value_binary vtminus"

definition kminus_safe where
  "kminus_safe = lift_value_binary vtminus_safe"

definition kmult where
  "kmult = lift_value_binary vtmult"

definition kmult_safe where
  "kmult_safe = lift_value_binary vtmult_safe"

definition kmod where
  "kmod = lift_value_binary vtmod"

definition kbytes_index where
  "kbytes_index = lift_value_binary vtbytes_index"

definition kbytes_and where
  "kbytes_and = lift_value_binary vtbytes_and"

definition kbytes_or where
  "kbytes_or = lift_value_binary vtbytes_or"

definition kbytes_xor where
  "kbytes_xor = lift_value_binary vtbytes_xor"

definition kbytes_not where
  "kbytes_not = lift_value_unary vtbytes_not"

definition kbytes_cast where
  "kbytes_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 kdnnot"

```

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'')"
  unfolding lift_op_monad_def by (simp add: execute_simps assms)

```

```

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"
  unfolding lift_op_monad_def by (simp add: execute_simps assms)

```

```

lemma lift_op_monad_simp3:
  assumes "execute lm s = Exception (e, s')"
  shows "execute (lift_op_monad op lm rm) s = Exception (e, s')"
  unfolding lift_op_monad_def by (simp add: execute_simps assms)

```

```

lemma lift_op_monad_simp4:
  assumes "execute lm s = NT"
  shows "execute (lift_op_monad op lm rm) s = NT"
  unfolding lift_op_monad_def by (simp add: execute_simps assms)

```

```

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'"
  unfolding lift_op_monad_def by (simp add: execute_simps assms)

```

```

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'')"
  unfolding equals_monad_def by (rule lift_op_monad_simp1[OF assms])

```

```

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"

```

```

unfolding equals_monad_def by (rule lift_op_monad_simp2[OF assms])

lemma equals_monad_simp3[execute_simps]:
  assumes "execute lm s = Exception (e, s'"
    shows "execute (equals_monad lm rm) s = Exception (e, s'"
  unfolding equals_monad_def by (rule lift_op_monad_simp3[OF assms])

lemma equals_monad_simp4[execute_simps]:
  assumes "execute lm s = NT"
    shows "execute (equals_monad lm rm) s = NT"
  unfolding equals_monad_def by (rule lift_op_monad_simp4[OF assms])

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'"
  unfolding equals_monad_def by (rule lift_op_monad_simp5[OF assms])

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 "<v" 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 kbytes_or"
```

```
definition bytes_xor_monad where
  "bytes_xor_monad = lift_op_monad kbytes_xor"
```

```
definition bytes_not_monad where
  "bytes_not_monad = lift_unary_monad kbytes_not"
```

```
definition bytes_cast_monad where
  "bytes_cast_monad m = lift_unary_monad (kbytes_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.Callldata (Some p) ⇒ cf (Location p) (Offset p)
            | kdata.Callldata 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'"
  using assms unfolding cond_monad_def
  by (cases "execute (equals_monad bm true_monad) s") (auto simp add: execute_simps split:if_split_asm)

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"
  using assms unfolding cond_monad_def
  by (cases "execute (equals_monad bm true_monad) s") (auto simp add: execute_simps split:if_split_asm)

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'"
  unfolding cond_monad_def by (simp add: execute_simps assms)

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'"
  unfolding cond_monad_def by (simp add: execute_simps assms kdbool_def)

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'"
  unfolding cond_monad_def by (simp add: execute_simps assms)

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"
  unfolding cond_monad_def by (simp add: execute_simps assms)

```

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.Callldata (Some (|Location=p, Offset=xs|))) =
      stack_disjoint i
        (K (throw Err))
        (λp'. option_disjoint
          (λs. state.Callldata s $$ p ≧≧ clookup xs)
          (λcd. memory_update_monad (mupdate is ∘ (read_calldata_memory cd p'))))
        (K (K (throw Err)))
        (modify (stack_update i (kdata.Callldata (Some (|Location=p, Offset=xs|)))) ≧≧ K (return Empty))
        (λp' xs'. option_disjoint
          (λs. state.Callldata s $$ p ≧≧ clookup (xs @ is))
          (λcd. storage_update_monad xs' is (K (read_calldata_storage cd)) p'))
        (throw Err)"
    | "assign_stack i is (rvalue.Callldata 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)"
 by (simp add: monotoneI sm_ordI)

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

proof (rule monotoneI)

fix xa::"'a ⇒ ('b, 'c, 'd) state_monad"

and ya::"'a ⇒ ('b, 'c, 'd) state_monad"

assume "sm.le_fun xa ya"

then show "sm_ord (K (xa x) y) (K (ya x) y)" using K.elims fun_ord_def by metis

qed

```

lemma lift_op_monad_mono:
  assumes "mono_sm A" and "mono_sm B"
  shows "mono_sm ( $\lambda f. \text{lift\_op\_monad op (A f) (B f)}$ )"
  unfolding lift_op_monad_def
proof (rule bind_mono[OF assms(1)])
  fix lv
  show "mono_sm ( $\lambda f. B f \ggg (\lambda rv. \text{option Err (K (op lv rv)) \ggg return}$ )")
  proof (rule bind_mono[OF assms(2)])
    fix rv
    show "mono_sm ( $\lambda f. \text{option Err (K (op lv rv)) \ggg return}$ )"
    proof (rule bind_mono)
      show "mono_sm ( $\lambda f. \text{option Err (K (op lv rv))}$ )" using option_monad_mono[of Err "K (op lv rv)"] by simp
    next
      fix y::('x::address) rvalue
      show "mono_sm ( $\lambda f. \text{return y}$ )" by (simp add: mono)
    qed
  qed
qed

```

```

lemma equals_monad_mono[mono]:
  assumes "mono_sm A" and "mono_sm B"
  shows "mono_sm ( $\lambda f. \text{equals\_monad (A f) (B f)}$ )"
  unfolding equals_monad_def by (rule lift_op_monad_mono[OF assms])

```

```

lemma cond_mono [partial_function_mono, mono]:
  assumes "mono_sm A" and "mono_sm B" and "mono_sm C"
  shows "mono_sm ( $\lambda f. \text{cond\_monad (A f) (B f) (C f)}$ )"
  unfolding cond_monad_def
proof (rule bind_mono)
  show "mono_sm ( $\lambda f. \text{equals\_monad (A f) true\_monad}$ )"
  proof (rule equals_monad_mono[OF assms(1)])
    show "mono_sm ( $\lambda f. \text{true\_monad}$ )" by (simp add: mono)
  qed
next
  fix b
  show "mono_sm ( $\lambda f. \text{if b = kdbool True then B f else if b = kdbool False then C f else throw Err}$ )"
  by (rule Partial_Function.if_mono[OF assms(2)], rule Partial_Function.if_mono[OF assms(3)]) (rule throw_monad_mono)
qed

```

```

partial_function (sm) while_monad :: "('a::address) expression_monad  $\Rightarrow$  ('a::address) expression_monad  $\Rightarrow$  ('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 $\text{while_monad } ?c \ ?m = \text{IF } ?c \ \text{THEN } ?m \ggg K (\text{while_monad } ?c \ ?m) \ \text{ELSE return Empty}$
- A general induction rule $\llbracket \text{sm.admissible } (\lambda \text{while_monad}. ?P (\text{curry while_monad})); ?P (\lambda \text{while_monad} c. \text{empty_result}); \bigwedge f. ?P f \implies ?P (\lambda a b. \text{IF } a \ \text{THEN } b \ggg K (f \ a \ b) \ \text{ELSE return Empty}) \rrbracket \implies ?P \ \text{Solidity.while_monad}$
- An induction rule for partial correctness $\llbracket \bigwedge \text{while_monad } c \ m \ ca \ ma. \llbracket \bigwedge a \ b \ h \ r. \text{effect (while_monad } a \ b) \ h \ r \implies ?P \ a \ b \ h \ r; \text{effect (IF } c \ \text{THEN } m \ggg K (\text{while_monad } c \ m) \ \text{ELSE return Empty}) } ca \ ma \rrbracket \implies ?P \ c \ m \ ca \ ma; \text{effect (Solidity.while_monad } ?c \ ?m) \ ?h \ ?r \rrbracket \implies ?P \ ?c \ ?m \ ?h \ ?r$

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 ical1_mono[mono]:
  assumes "mono_sm (λx. m x)"
  shows "mono_sm (λx. ical1 (m x))"
  unfolding ical1_def using assms by (simp add:mono)

```

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))"
  unfolding ecall_def using assms by (simp add:mono)

```

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 " $\bigwedge r. \text{effect } x \text{ s r} \implies \text{post } s \text{ r I (K True) P}$ "
    shows "post s r I (K True) P"
  using assms(1) unfolding post_def effect_def exc_def
  apply (auto simp add:execute_simps) using assms(2) unfolding effect_def post_def
  apply (metis (no_types, lifting) case_prod_conv old.sum.simps(5) prod.split_sel
    result.case_eq_if result.disc(2,3) result.sel(1))
  using assms(2) unfolding effect_def post_def
  apply (smt (verit, del_insts) case_prod_beta old.sum.simps(5) result.case_eq_if
    result.disc(2,3) result.sel(1) snd_eqD)
  by (simp add: inv_state_def)

lemma post_exc_false:
  assumes "effect (exc x) s r"
    and " $\bigwedge r. \text{effect } x \text{ s r} \implies \text{post } s \text{ r I (K False) P}$ "
    shows "post s r I (K False) P"
  using assms(1) unfolding post_def effect_def exc_def
  apply (auto simp add:execute_simps) using assms(2) unfolding effect_def post_def
  apply (smt (verit, del_insts) case_prod_beta ex.case ex.exhaust fst_def is_Normal_def
    old.sum.simps(5) prod.collapse result.case_eq_if result.disc(2) result.disc(3) result.distinct_disc(1)
    result.sel(1) snd_def)
  using assms(2) unfolding effect_def post_def
  apply (smt (verit, del_insts) case_prod_beta ex.case ex.exhaust old.sum.simps(5) prod.collapse
    result.case_eq_if result.disc(2) result.disc(3) result.sel(1))
  by (metis (no_types, lifting) K.simps assms(2) effect_def inv_state_def old.sum.simps(6) post_def
    result.case_eq_if result.collapse(2) result.distinct(1) result.distinct(5) split_beta)

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"
  using assms unfolding post_def effect_def
  apply (auto simp add: execute_simps)
  unfolding exc_def apply (simp add:execute_simps)
  by (metis (mono_tags, lifting) ex.exhaust result.case_eq_if result.disc(4) result.disc(6)
    result.sel(2) snd_conv split_beta)

end

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

definition transfer_monad::
  "('d  $\Rightarrow$  'a expression_monad)  $\Rightarrow$  ('a::address) expression_monad  $\Rightarrow$  ('a::address) expression_monad  $\Rightarrow$ 
  ('a::address) expression_monad"
  ("⟨transfer⟩")
where
  "transfer_monad call am vm =
  do {
    ak  $\leftarrow$  am;
    av  $\leftarrow$  readValue ak;
    a  $\leftarrow$  readAddress av;
    vk  $\leftarrow$  vm;
    vv  $\leftarrow$  readValue vk;
    v  $\leftarrow$  readSint vv;
    assert Err ( $\lambda$ s. Balances s this  $\geq$  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)"
  unfolding transfer_monad_def
  by (auto intro!: mono)

```

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"
  unfolding array_def
  by simp

```

```

lemma fold_map_write_replicate_length:
  assumes "fold_map Memory.write (replicate n (adata.Value v)) m = (x1, x2)"
  shows "length x1 = n"
  using assms
proof (induction n arbitrary: x1 m)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.premis obtain x1a
  where *: "fold_map Memory.write (replicate n (adata.Value v)) (m @ [mdata.Value v]) = (x1a, x2)"
        and **: "x1 = length m # x1a"
  by (auto simp add: array_def length_append_def split:prod.split_asm)
  then show ?case
proof (cases "n = 0")
  case True
  then show ?thesis using Suc.premis(1) by (auto split:prod.split_asm)
next
  case False
  then show ?thesis using * Suc by (auto simp add: length_append_def)
qed

```

```

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))"
using assms
proof (induction n arbitrary: x1 m x)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from Suc.prem1
  obtain x1a
  where *: "fold_map Memory.write (replicate n (adata.Value (Uint 0))) (m @ [mdata.Value (Uint 0)])
= (x1a, x2)"
    and **: "x1 = length m # x1a"
  by (simp add: length_append_def split:prod.split_asm)
  then show ?case
  proof (cases "n = 0")
    case True
    then show ?thesis using Suc.prem1,2 by (auto simp add: length_append_def)
  next
    case False
    then show ?thesis
    proof (cases "x = 0")
      case True
      moreover from False
      have "(replicate n (adata.Value (Uint 0))) ≠ []" by auto
      then have "sprefix (m @ [mdata.Value (Uint 0)]) x2"
      using write_fold_map_sprefix[of "(replicate n (adata.Value (Uint 0)))" " (m @ [mdata.Value
(Uint 0)])"]
      unfolding sprefix_def using * by simp
      ultimately show ?thesis unfolding sprefix_def
      using ** sprefix_def by (auto simp add: array_def length_append_def split:prod.split_asm)
    next
      case _ : False
      moreover have "x1a ! (x - 1) < length x2 ∧ (∃ix. x2 ! (x1a ! (x - 1)) = mdata.Value (Uint ix))"
      using Suc.IH[OF *] Suc.prem2 False by simp
      ultimately show ?thesis
      using * Suc.prem1 by (auto simp add: length_append_def)
    qed
  qed
qed

```

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 -
  from assms obtain x1a x2a
  where *: "fold_map Memory.write (replicate (unat si) (adata.Value (Uint 0))) [] = (x1a, x2a)"
    and "x1 = length x2a"
    and "x2 = x2a @ [mdata.Array x1a]"
  by (simp add: array_def length_append_def split:prod.split_asm)
  moreover have "(∀i < length x1a. (x1a ! i) < length x2a ∧ (∃ix. x2 ! (x1a ! i) = mdata.Value (Uint
ix)))"
  proof (rule allI, rule impI)
    fix i assume "i < length x1a"
    moreover have "length x1a = unat si" using fold_map_write_replicate_length[OF *] by simp
    ultimately show "(x1a ! i) < length x2a ∧ (∃ix. x2 ! (x1a ! i) = mdata.Value (Uint ix))"
    using fold_map_write_replicate_value[OF *, of i]
    by (simp add: <x2 = x2a @ [mdata.Array x1a]> nth_append_left)
  qed
  ultimately show ?thesis by auto
qed

```

lemma mupdate_array_typing_value:

```

  assumes "state.Memory sa ! m1 = 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] (m1, mdata.Value (Uint x), state.Memory sa) = Some yg"
    shows "∃ma0. yg ! m1 = mdata.Array ma0
    ∧ (∀i<length ma0. (ma0 ! i) < length yg ∧ (∃ix. yg ! (ma0 ! i) = mdata.Value (Uint ix)))"
proof -
  from assms have "ma0 ! unat xa ≠ m1"
  proof -
    from assms(1,2,3) obtain ix
    where "(ma0 ! (unat xa)) < length (state.Memory sa)"
    and "(state.Memory sa ! (ma0 ! (unat xa))) = mdata.Value (Uint ix)"
    by (auto simp add: case_memory_def nth_safe_def split:if_split_asm)
    then show ?thesis using assms(1) by auto
  qed
then have "yg ! m1 = mdata.Array ma0"
  using assms(1,3)
  by (auto simp add: case_memory_def nth_safe_def list_update_safe_def split:if_split_asm)
moreover have "∀i<length ma0. (ma0 ! i) < length yg ∧ (∃ix. yg ! (ma0 ! i) = mdata.Value (Uint
ix))"
proof (rule allI, rule impI)
  fix i assume "i < length ma0"
  show "(ma0 ! i) < length yg ∧ (∃ix. yg ! (ma0 ! i) = mdata.Value (Uint ix))"
  proof (cases "ma0 ! i = ma0 ! unat xa")
    case True
    then show ?thesis using assms(1,3)
    by (auto simp add: case_memory_def nth_safe_def list_update_safe_def split:if_split_asm)
  next
    case False
    then show ?thesis using <i < length ma0> assms(1,2,3)
    by (auto simp add: case_memory_def nth_safe_def list_update_safe_def split:if_split_asm)
  qed
qed
ultimately show ?thesis by blast
qed

```

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 o 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 := fmpud 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 (callldata_update i c o stack_update i (kdata.Callldata (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 <
val it = term <DArray (TValue (TBytes 1))>
>

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_file Utils.ML
ML_file Data.ML
ML_file Specification.ML
ML_file Invariant.ML
ML_file Verification.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"
  unfolding mapping_def ..
```

```
lemma Value_vt [wpsimps]:
  assumes "storage_data.Value x = v"
  shows "storage_data.vt v = x"
  using assms by auto
```

Kdata

```
lemma kdbool_simp [wpsimps]:
  "kdbool x = Value (Bool x)"
  unfolding kdbool_def by simp
```

```
lemma kdSint_simp [wpsimps]:
  "kdSint x = Value (Uint x)"
  unfolding kdSint_def by simp
```

```
lemma kdBytes_simp [wpsimps]:
  "kdBytes xs = Value (Bytes xs)"
  unfolding kdBytes_def by simp
```

```
lemma kdAddress_simp [wpsimps]:
  "kdAddress x = Value (Address x)"
  unfolding kdAddress_def by simp
```

```
lemma kadminus [wpsimps]:
  "kadminus (rvalue.Value (Uint 1)) (rvalue.Value (Uint r)) = Some (rvalue.Value (Uint (1 - r)))"
  unfolding kadminus_def vtminus_def by simp
```

```
lemma kadminus_safe [wpsimps]:
  assumes "r ≤ 1"
  shows "kadminus_safe (rvalue.Value (Uint 1)) (rvalue.Value (Uint r)) = Some (rvalue.Value (Uint (1 - r)))"
```

```

unfolding kdminus_safe_def using assms by (simp add: vtminus_safe.simps)

lemma kdminus_safe_dest[wpdrules]:
  assumes "kdminus_safe (rvalue.Value (Uint 1)) (rvalue.Value (Uint r)) = Some ya"
  shows "r ≤ 1 ∧ ya = rvalue.Value (Uint (1 - r))"
  using assms unfolding kdminus_safe_def by (simp split:if_split_asm add:vtminus_safe.simps)

lemma kdminus_storage[wpsimps]:
  "kdminus (rvalue.Storage x) z = None"
  unfolding kdminus_def vtminus_def by simp

lemma kdplus[wpsimps]:
  "kdplus (rvalue.Value (Uint 1)) (rvalue.Value (Uint r)) = Some (rvalue.Value (Uint (1 + r)))"
  unfolding kdplus_def vtplus_def by simp

lemma kdplus_safe[wpsimps]:
  assumes "unat 1 + unat r < 2256"
  shows "kdplus_safe (rvalue.Value (Uint 1)) (rvalue.Value (Uint r)) = Some (rvalue.Value (Uint (1 + r)))"
  unfolding kdplus_safe_def using assms by (simp add:vtplus_safe.simps)

lemma kdplus_safe_dest[wpdrules]:
  assumes "kdplus_safe (rvalue.Value (Uint 1)) (rvalue.Value (Uint r)) = Some ya"
  shows "unat 1 + unat r < 2256 ∧ ya = rvalue.Value (Uint (1 + r))"
  using assms unfolding kdplus_safe_def by (simp split:if_split_asm add:vtplus_safe.simps)

lemma kdmult[wpsimps]:
  "kdmult (rvalue.Value (Uint 1)) (rvalue.Value (Uint r)) = Some (rvalue.Value (Uint (1 * r)))"
  unfolding kdmult_def vtmult_def by simp

lemma kdmult_safe[wpsimps]:
  assumes "unat 1 * unat r < 2256"
  shows "kdmult_safe (rvalue.Value (Uint 1)) (rvalue.Value (Uint r)) = Some (rvalue.Value (Uint (1 * r)))"
  unfolding kdmult_safe_def using assms by (simp add:vtmult_safe.simps)

lemma kdmult_safe_dest[wpdrules]:
  assumes "kdmult_safe (rvalue.Value (Uint 1)) (rvalue.Value (Uint r)) = Some ya"
  shows "unat 1 * unat r < 2256 ∧ ya = rvalue.Value (Uint (1 * r))"
  using assms unfolding kdmult_safe_def by (simp split:if_split_asm add:vtmult_safe.simps)

```

Updates

```

lemma stack_stack_update_diff[wpsimps]:
  assumes "i ≠ i'"
  shows "Stack (stack_update i x s) $$ i' = Stack s $$ i'"
  using assms unfolding stack_update_def by simp

lemma (in Contract) stack_storage_update[wpsimps]:
  "Stack (storage_update i x s) = Stack s"
  unfolding storage_update_def by simp

lemma stack_balances_update[wpsimps]:
  "Stack (balances_update i x s) = Stack s"
  unfolding balances_update_def by simp

lemma stack_calldata_update[wpsimps]:
  "Stack (calldata_update i x s) = Stack s"
  unfolding calldata_update_def by simp

lemma stack_update_eq[wpsimps]:
  "Stack (stack_update i x s) $$ i = Some x"
  unfolding stack_update_def by simp

```

```

lemma memory_balances_update[wpsimps]:
  "state.Memory (balances_update i x s) = state.Memory s"
  unfolding balances_update_def by simp

lemma memory_stack_update[wpsimps]:
  "state.Memory (stack_update i x s) = state.Memory s"
  unfolding stack_update_def by simp

lemma calldata_balances_update[wpsimps]:
  "state.Calldata (balances_update i x s) = state.Calldata s"
  unfolding balances_update_def by simp

lemma calldata_stack_update[wpsimps]:
  "state.Calldata (stack_update i x s) = state.Calldata s"
  unfolding stack_update_def by simp

lemma storage_stack_update[wpsimps]:
  "state.Storage (stack_update i v s) = state.Storage s"
  unfolding stack_update_def by simp

lemma storage_calldata_update[wpsimps]:
  "state.Storage (calldata_update i v s) = state.Storage s"
  unfolding calldata_update_def by simp

lemma storage_balances_update[wpsimps]:
  "state.Storage (balances_update i v s) = state.Storage s"
  unfolding balances_update_def by simp

lemma calldata_calldata_update[wpsimps]:
  "state.Calldata (calldata_update i v s) $$$ i = Some v"
  unfolding calldata_update_def by simp

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'"
  using assms unfolding storage_update_def by simp

lemma (in Contract) storage_update_eq[wpsimps]:
  "state.Storage (storage_update i x s) this i = x"
  unfolding storage_update_def by simp

lemma (in Contract) balances_storage_update[wpsimps]:
  "Balances (storage_update i' x s) = Balances s"
  unfolding storage_update_def by simp

lemma balances_stack_update[wpsimps]:
  "Balances (stack_update i' x s) = Balances s"
  unfolding stack_update_def by simp

lemma balances_balances_update_diff[wpsimps]:
  assumes "i ≠ i'"
  shows "Balances (balances_update i x s) i' = Balances s i'"
  using assms unfolding balances_update_def by simp

lemma balances_balances_update_same[wpsimps]:
  "Balances (balances_update i x s) i = x"
  unfolding balances_update_def by simp

```

6.1.2 Destruction rules

```

lemma some_some[wpdrules]:
  assumes "Some x = Some y"
  shows "x = y" using assms by simp

```

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"
unfolding wp_def by (cases "execute f s" rule:result_cases) (simp_all add: assms)
```

```
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"
using assms unfolding wp_def by (cases "execute f s" rule:result_cases) simp_all
```

```
lemma wp_simp1:
```

```
assumes "execute f s = Normal (r, s'"
shows "wp f P E s = P r s'"
unfolding wp_def by (cases "execute f s" rule:result_cases) (simp_all add: assms)
```

```
lemma wp_simp2:
```

```
assumes "execute f s = Exception (e, s'"
shows "wp f P E s = E e s'"
unfolding wp_def by (cases "execute f s" rule:result_cases) (simp_all add: assms)
```

```
lemma wp_simp3:
```

```
assumes "execute f s = NT"
shows "wp f P E s"
unfolding wp_def by (cases "execute f s" rule:result_cases) (simp_all add: assms)
```

```
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"
using assms by simp
```

```
lemma wpreturn[wprules]:
```

```
assumes "P x s"
shows "wp (return x) P E s"
unfolding wp_def using assms by (simp add: execute_simps)
```

```
lemma wpget[wprules]:
```

```
assumes "P s s"
shows "wp get P E s"
unfolding wp_def using assms by (simp add: execute_simps)
```

```
lemma wpbind[wprules]:
```

```
assumes "wp f (λa. (wp (g a) P E)) E s"
shows "wp (f >>= g) P E s"
proof (cases "execute f s")
case nf: (n a s')
then have **: "wp (g a) P E s'" using wp_def[of f "λa. wp (g a) P E"] assms by simp
show ?thesis
proof (cases "execute (g a) s'")
case ng: (n a' s'')
then have "P a' s'" using wp_def[of "g a" P] ** by simp
```

```

    moreover from nf ng have "execute (f  $\gg$  g) s = Normal (a', s'')" by (simp add: execute_simps)
    ultimately show ?thesis using wp_def by fastforce
next
  case (e e s'')
  then have "E e s''" using wp_def[of "g a" P] ** by simp
  moreover from nf e have "execute (f  $\gg$  g) s = Exception (e, s'')" by (simp add: execute_simps)
  ultimately show ?thesis using wp_def by fastforce
next
  case t
  with nf have "execute (f  $\gg$  g) s = NT" by (simp add: execute_simps)
  then show ?thesis using wp_def by fastforce
qed
next
  case (e e s')
  then have "E e s'" using wp_def[of f "\lambda a. wp (g a) P E"] assms by simp
  moreover from e have "execute (f  $\gg$  g) s = Exception (e, s')" by (simp add: execute_simps)
  ultimately show ?thesis using wp_def by fastforce
next
  case t
  then have "execute (f  $\gg$  g) s = NT" by (simp add: execute_simps)
  then show ?thesis using wp_def by fastforce
qed

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

lemma wp_lfold:
  assumes "P [] s"
  assumes "\^a list. xs = a#list  $\implies$  wp (a  $\gg$  (\lambda l. option Err (\lambda_. the_value l)  $\gg$  (\lambda l'. lfold list
 $\gg$  (\lambda ls. return (l' # ls)))))) P E s"
  shows "wp (lfold xs) P E s"
  using assms unfolding wp_def
  apply (cases xs)
  by (simp_all add: execute_simps)

lemma result_cases2[cases type: result]:
  fixes x :: "('a  $\times$  's, 'e  $\times$  's) result"
  obtains (n) a s e where "x = Normal (a, s)  $\vee$  x = Exception (e, s)"
    | (t) "x = NT"
proof (cases x)
  case (n a s)
  then show ?thesis using that by simp
next
  case (e e)
  then show ?thesis using that by fastforce
next
  case t
  then show ?thesis using that by simp
qed

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

lemma wpnewStack[wprules]:
  assumes "P Empty (s(|Stack := { $$ }|))"
  shows "wp newStack P E s"
  unfolding wp_def newStack_def using assms by (simp add: execute_simps)

lemma wpnewMemory[wprules]:
  assumes "P Empty (s(|Memory := []|))"

```

```

shows "wp newMemory P E s"
unfolding wp_def newMemory_def using assms by (simp add: execute_simps)

lemma wpnewCallldata[wprules]:
  assumes "P Empty (s(Callldata := {$$}))"
  shows "wp newCallldata P E s"
  unfolding wp_def newCallldata_def using assms by (simp add: execute_simps)

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"
  unfolding lift_op_monad_def using assms by (rule wprules)

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"
  unfolding equals_monad_def using assms by (rule wp_lift_op_monad)

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"
  unfolding less_monad_def using assms by (rule wp_lift_op_monad)

lemma wp_mod_monad[wprules]:
  assumes "wp lm (λa. wp (rm ≫= (λrv. option Err (K (kdmmod a rv)) ≫= return)) P E) E s"
  shows "wp (mod_monad lm rm) P E s"
  unfolding mod_monad_def using assms by (rule wp_lift_op_monad)

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"
  unfolding minus_monad_def using assms by (rule wp_lift_op_monad)

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"
  unfolding minus_monad_safe_def using assms by (rule wp_lift_op_monad)

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"
  unfolding plus_monad_def using assms by (rule wp_lift_op_monad)

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"
  unfolding plus_monad_safe_def using assms by (rule wp_lift_op_monad)

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"
  unfolding mult_monad_def using assms by (rule wp_lift_op_monad)

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"
  unfolding mult_monad_safe_def using assms by (rule wp_lift_op_monad)

lemma wp_bool_monad[wprules]:
  assumes "P (kdbool b) s"
  shows "wp (bool_monad b) P E s"
  unfolding bool_monad_def using assms by (simp add: wprules)

lemma wp_true_monad[wprules]:

```

```

assumes "P (kdbool True) s"
shows "wp true_monad P E s"
unfolding true_monad_def using assms by (rule wp_bool_monad)

lemma wp_false_monad[wprules]:
  assumes "P (kdbool False) s"
  shows "wp false_monad P E s"
  unfolding false_monad_def using assms by (rule wp_bool_monad)

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"
  unfolding or_monad_def kdor_def using assms by (rule wp_lift_op_monad)

lemma wp_sint_monad[wprules]:
  assumes "P (kdSint x) s"
  shows "wp (sint_monad x) P E s"
  unfolding sint_monad_def using assms by (simp add: wprules)

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"
  unfolding bytes_monad_def using assms by (simp add: wprules)

lemma (in Method) wp_value_monad[wprules]:
  assumes "P (kdSint msg_value) s"
  shows "wp value_monad P E s"
  unfolding value_monad_def using assms by (rule wp_sint_monad)

lemma (in Method) wp_stamp_monad[wprules]:
  assumes "P (kdSint timestamp) s"
  shows "wp block_timestamp_monad P E s"
  unfolding block_timestamp_monad_def using assms by (rule wp_sint_monad)

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"
  unfolding cond_monad_def
  apply (rule wprules)+ by (rule assms)

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"
  unfolding assert_monad_def
  using assms by simp

lemma wption[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 (cases "f s")
  case None
  then show ?thesis unfolding option_def wp_def using assms(2) by (simp add:execute_simps)
next
  case (Some a)
  then show ?thesis unfolding option_def wp_def using assms(1) by (simp add:execute_simps)
qed

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"
  unfolding lift_unary_monad_def apply (rule wprules)+ by (rule assms)

```

```

lemma wp_not_monad[wprules]:
  assumes "wp lm ( $\lambda a. wp (option Err (K (kdnnot a)) \gg= return) P E) E s"$ 
  shows "wp (not_monad lm) P E s"
  unfolding not_monad_def using assms by (rule wp_lift_unary_monad)

lemma wp_address_monad[wprules]:
  assumes "P (kdAddress a) s"
  shows "wp (address_monad a) P E s"
  unfolding address_monad_def by (simp add: wprules assms)

lemma (in Method) wp_sender_monad[wprules]:
  assumes "P (kdAddress msg_sender) s"
  shows "wp sender_monad P E s"
  unfolding sender_monad_def using assms by (rule wp_address_monad)

lemma wp_require_monad[wprules]:
  assumes "wp ( $x \gg= (\lambda v. if v = rvalue.Value (Bool True) then return Empty else throw Err) P E s"$ 
  shows "wp (require_monad x) P E s"
  unfolding require_monad_def using assms by (simp add: wpsimps)

lemma (in Contract) wp_storeLookup[wprules]:
  assumes "wp (lfold es)
    ( $\lambda a. wp (option Err (\lambda s. slookup a (state.Storage s this i)) \gg=
      (\lambda sd. if storage_data.is_Value sd then return (rvalue.Value (storage_data.vt sd)) else
        return (rvalue.Storage (Some (Location=i, Offset=a)))) P E)
    E s"$ 
  shows "wp (storeLookup i es) P E s"
  unfolding storeLookup_def by (rule wprules | auto simp add: assms split:if_split)+

lemma wpassert[wprules]:
  assumes "t s  $\implies$  wp (return ()) P E s"
  and " $\neg t s \implies wp (throw x) P E s"$ 
  shows "wp (assert x t) P E s"
  unfolding wp_def apply (cases "execute (assert x t) s") apply (auto simp add: execute_simps)
  apply (metis assms(1) assms(2) execute_assert(1) execute_assert(2) wp_simp1)
  by (metis assms(1) assms(2) execute_assert(1) execute_assert(2) wp_simp2)

lemma wp_bool[wprules]:
  "wp (bool_monad b) ( $\lambda a \_ . a = kdbool b) (K x) s"$ 
  unfolding bool_monad_def
  by (simp add: wprules)

lemma wpskip[wprules]:
  assumes "P Empty s"
  shows "wp skip_monad P E s"
  unfolding skip_monad_def using assms by vcg

lemma effect_bind:
  assumes "effect (m  $\gg= (\lambda x. n x) ss r"$ 
  and "execute m ss = Normal (a2, s)"
  shows "effect (n a2) s r"
  using assms unfolding cond_monad_def effect_def bind_def execute_create by simp

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"
  using assms unfolding cond_monad_def
  by (metis (no_types, lifting) assms(1) execute_cond_monad_simp1 effect_def)

lemma wpwhile:

```

```

assumes "\s. iv s
        ==> wp (equals_monad c true_monad)
            (\a s. (a = kdbool True ==> wp m (K iv) E s) \
                (a = kdbool False ==> P Empty s) \
                (a \neq kdbool False \wedge a \neq kdbool True ==> E Err s))
            E s"
and "iv s"
shows "wp (while_monad c m) P E s"
proof (cases "execute (while_monad c m) s" rule: result_cases2)
case (n a s' ex)
then obtain r where effect_while:"effect (while_monad c m) s r" unfolding effect_def by auto
show ?thesis using assms
proof (induction rule: while_monad.raw_induct[OF _ effect_while])
case a: (1 while_monad' c m ss sn)
have "wp (cond_monad c (bind m (K (while_monad c m))) (return Empty)) P E ss"
proof (rule wpI)
fix a s'
assume "execute (cond_monad c (bind m (K (while_monad c m))) (return Empty)) ss = Normal (a, s')"
then show "P a s'"
proof (rule execute_cond_monad_normal_E)
fix s''
assume "execute (equals_monad c true_monad) ss = Normal (kdbool True, s'')"
and "execute (m \gg= K (while_monad c m)) s'' = Normal (a, s'')"
then have execute_equals: "execute (equals_monad c true_monad) ss = Normal (kdbool True, s'')"
and "execute (m \gg= K (while_monad c m)) s'' = Normal (a, s'')" by simp+
from this(2) show "P a s'"
proof (rule execute_bind_normal_E)
fix s''' x
assume execute_m: "execute m s'' = Normal (x, s'''"
and execute_while: "execute (K (while_monad c m) x) s''' = Normal (a, s'''"
moreover from a(3)[OF a(4)] have "wp m (K iv) E s'''" using execute_equals unfolding wp_def
by simp
ultimately have "iv s'''" unfolding wp_def by (cases "execute m s'''" (simp)+
moreover from a(2) obtain sn where "effect (while_monad' c m) s''' sn"
proof -
from effect_cond_monad[OF a(2) execute_equals]
have "effect (m \gg= K (while_monad' c m)) s''' sn" by simp
with effect_bind show ?thesis using that execute_m by fastforce
qed
ultimately have "wp (while_monad c m) P E s'''" using a(1)[OF _ a(3), where ?h=s'''" by
simp
with execute_while show "P a s'" unfolding wp_def by simp
qed
next
fix s''
assume execute_equals: "execute (equals_monad c true_monad) ss = Normal (kdbool False, s'')"
and "execute (return Empty) s'' = Normal (a, s'')"
then have "s'' = s'" using execute_returnE by meson
moreover from a(3)[OF a(4)] have "P Empty s'''" using execute_equals unfolding wp_def by simp
ultimately show "P a s'" by (metis <execute (return Empty) s'' = Normal (a, s')>
execute_returnE(1))
qed
next
fix x s'
assume "execute (Solidity.cond_monad c (m \gg= K (while_monad c m)) (return Empty)) ss = Exception
(x, s'"
then show "E x s'"
proof (rule execute_cond_monad_exception_E)
assume "execute (equals_monad c true_monad) ss = Exception (x, s'"
then show "E x s'" using a(3)[OF a(4)] unfolding wp_def by simp
next
fix a
assume "execute (equals_monad c true_monad) ss = Normal (a, s'"
and "a \neq kdbool True \wedge a \neq kdbool False \wedge x = Err"

```

```

    then show "E x s'" using a(3)[OF a(4)] unfolding wp_def by simp
  next
    fix s''
    assume execute_equals: "execute (equals_monad c true_monad) ss = Normal (kdbool True, s'')"
    and "execute (m ≧≧ K (while_monad c m)) s'' = Exception (x, s'')"
    then have "execute (m ≧≧ K (while_monad c m)) s'' = Exception (x, s'')" by simp
    then show "E x s'"
    proof (rule execute_bind_exception_E)
      assume "execute m s'' = Exception (x, s'')"
      then show "E x s'" using a(3)[OF a(4)] execute_equals unfolding wp_def by simp
    next
      fix a s''''
      assume execute_m: "execute m s'' = Normal (a, s'''' )"
      and execute_while: "execute (K (while_monad c m) a) s'''' = Exception (x, s'')"
      moreover from a(3)[OF a(4)] have "wp m (K iv) E s'''' " using execute_equals unfolding wp_def
    by simp
    ultimately have "iv s'''' " unfolding wp_def by (cases "execute m s'''' ") (simp)+
    moreover from a(2) obtain sn where "effect (while_monad' c m) s'''' sn"
    proof -
      from effect_cond_monad[OF a(2) execute_equals]
      have "effect (m ≧≧ K (while_monad' c m)) s'''' sn" by simp
      with effect_bind show ?thesis using that execute_m by fastforce
    qed
    ultimately have "wp (while_monad c m) P E s'''' " using a(1)[OF _ a(3), where ?h=s'''' ] by
  simp
  with execute_while show "E x s'" unfolding wp_def by simp
  qed
next
  fix s''
  assume "execute (equals_monad c true_monad) ss = Normal (kdbool False, s'')"
  and "execute (return Empty) s'' = Exception (x, s'')"
  then show "E x s'" by (simp add:execute_return)
  qed
  then show "wp (while_monad c m) P E ss" by (subst while_monad.simps)
  qed
next
  case t
  then show ?thesis unfolding wp_def by simp
  qed

lemma wp_applyf[wprules]:
  assumes "P (f s) s"
  shows "wp (applyf f) P E s"
  unfolding applyf_def get_def return_def wp_def using assms by (auto simp add:wpsimps execute_simps)

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"
  unfolding wp_def apply (cases x, auto) apply (fold wp_def) by (simp add:assms)+

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"
  unfolding wp_def apply (cases a, auto) apply (fold wp_def) by (simp add:assms)+

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

```

```

unfolding init_def wp_def kinit_def using assms by (auto simp add:wpsimps execute_simps)

lemma wp_decl[wprules]:
  assumes "wp (init (Solidity.default t) i) P E s"
  shows "wp (decl t i) P E s"
  unfolding decl_def using assms by simp

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"
  unfolding write_def wp_def using assms by (auto simp add:wpsimps execute_simps split: prod.split)

lemma wp_sinit[wprules]:
  assumes "P Empty (stack_update i (kdata.Storage None) s)"
  shows "wp (sinit i) P E s"
  unfolding sinit_def wp_def using assms by (auto simp add:wpsimps execute_simps)

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"
  unfolding wp_def apply (case_tac t) using assms by (auto simp add:wpsimps sdecl.simps execute_simps
wp_def)

lemma (in Contract) wp_initStorage[wprules]:
  assumes "P Empty (storage_update i v s)"
  shows "wp (initStorage i v) P E s"
  unfolding initStorage_def wp_def using assms by (auto simp add:wpsimps execute_simps)

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"
  unfolding init_balance_def wp_def using assms by (auto simp add:wpsimps execute_simps)

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"
  unfolding init_balance_np_def wp_def using assms by (auto simp add:wpsimps execute_simps)

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"
  unfolding cinit_def wp_def using assms by (auto simp add:wpsimps execute_simps)

lemma (in Contract) wp_assign_stack_monad[wprules]:
  assumes "wp m (\lambda. wp (lfold is >>= (\lambda is. assign_stack i is a >>= (\lambda_. return Empty)))) P E) E s"
  shows "wp (assign_stack_monad i is m) P E s"
  unfolding assign_stack_monad_def apply (rule wprules) using assms by simp

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"
  unfolding storage_update_monad_def by (rule wprules | simp add: assms)+

lemma (in Contract) wp_assign_storage1[wprules]:
  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"
    using assms by simp

```

```

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"
  using assms by simp

```

```

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"
  unfolding assign_storage_monad_def apply (rule wprules) using assms by simp

```

```

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"
  unfolding stackLookup_def apply (vcg) using assms by simp

```

```

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"
  unfolding keccak256_monad_def using assms by (rule wprules)+

```

```

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"
  unfolding transfer_monad_def apply (rule wprules)+ by (rule assms)

```

```

lemma wp_readValue[wprules]:
  assumes "P (storage_data.vt yp) s"
  shows "wp (readValue (rvalue.Value (storage_data.vt yp))) P E s"
  unfolding wp_def readValue.simps by (simp add:execute_return assms)

```

```

lemma wp_readAddress[wprules]:

```

```

assumes "P yp s"
shows "wp (readAddress (Address yp)) P E s"
unfolding wp_def readAddress.simps by (simp add:execute_return assms)

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 $$ i = Some (kdata.Calldata None) ==> wp cp P E s"
  shows "wp (stack_disjoint i kf mf cf cp sf sp) P E s"
unfolding wp_def stack_disjoint_def
apply (simp add:execute_simps applyf_def get_def return_def bind_def)
apply (cases "Stack s $$ i")
apply (auto simp add:execute_simps)
defer apply (case_tac a)
apply (fold wp_def) using assms
by (auto simp add:wprules)

lemma execute_normal:
  assumes "execute x s = Normal (a, b)"
  shows "effect x s (Inl (a,b))" using assms unfolding effect_def by simp

lemma execute_exception:
  assumes "execute x s = Exception (a, b)"
  shows "effect x s (Inr (a,b))" using assms unfolding effect_def by simp

lemma (in Contract) inv_wp:
  assumes "effect m s r"
    and "wp m (K x) (K y) s"
  shows "inv r x y"
using assms unfolding inv_def effect_def wp_def apply (cases "execute m s") by auto

lemma (in Contract) post_wp:
  assumes "effect m s r"
    and "wp m (\r s'. P s r s' ^ inv_state Is s') (K (inv_state Ie)) s"
  shows "post s r Is Ie P"
using assms unfolding post_def effect_def wp_def apply (cases "execute m s") by auto

lemma (in Contract) wp_storeArrayLength[wprules]:
  assumes "wp (ifold xs)
(\a. wp (option Err (\s. slookup a (state.Storage s this v)) >=
(\sd. storage_disjoint sd (K (throw Err)) (\sa. 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"
unfolding storeArrayLength_def apply vcg using assms by simp

lemma (in Contract) wp_arrayLength[wprules]:
  assumes "wp (ifold xs)
(\a. wp (stack_disjoint v (K (throw Err))
(\p. option Err (\s. mlookup (state.Memory s) a p) >=
(\l. option Err (\s. state.Memory s $ l) >=
(\md. if mdata.is_Array md
then return (rvalue.Value (Uint (word_of_nat (length (mdata.ar
md)))))) else throw Err)))
(\p xs.
option Err (\s. state.Calldata s $$ p >= clookup (xs @ a)) >=
(\sd. 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"
  unfolding arrayLength_def apply vcg using assms by simp

```

```

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"
  unfolding storeArrayLength_def apply vcg using assms apply simp apply vcg done

```

```

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"
  using assms apply (cases sd) by (simp add:wpsimps)+

```

```

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"
  unfolding allocate_def apply vcg using assms by simp

```

```

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"
  unfolding create_memory_array_def apply vcg using assms by simp

```

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"
  using assms unfolding wp_def effect_def post_def inv_state_def
  by (cases "execute (c x) s") (auto)

```

```

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 $$ i = Some (kdata.Calldata None)  $\implies$  E Err s"
and " $\bigwedge$ aa. Stack s $$ i = Some (kdata.Storage (Some aa))  $\implies$ 
  wp (storage_update_monad (Offset aa) is (K (storage_data.Value v)) (Location aa)) P E s"
and " $\bigwedge$ x4. Stack s $$ i = Some (kdata.Value x4)  $\implies$ 
  wp (modify (stack_update i (kdata.Value v))  $\gg=$  ( $\lambda$ a. return Empty)) P E s"
and " $\bigwedge$ a. Stack s $$ i = Some (kdata.Calldata (Some a))  $\implies$  E Err s"
shows "wp (assign_stack i is (rvalue.Value v)) P E s"
apply (vcg | auto simp add:assms stack_disjoint_def)+
using assms apply blast
by (vcg | auto simp add:assms stack_disjoint_def)+
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_Natural" "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
by standard (simp add: null_def)

```

```

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
.

```

```

global_interpretation keccak256: Keccak256 id
defines keccak256_keccak256_monad = keccak256.keccak256_monad
by standard simp

```

7.2 States (Unit_Tests)

definition *emptyState*::"aspace state" **where**

```

"emptyState =
(|
    state.Memory = [],
    state.Callldata = fmemory,
    state.Storage = (λ_ . undefined),
    state.Stack = fmemory,
    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))"
by (normalization,simp)

```

```

lemma "P (execute false_monad emptyState)
  = P (Normal (rvalue.Value (Bool False),emptyState))"
by (normalization,simp)

```

```

lemma "P (execute (sint_monad 5) emptyState)
  = P (Normal (rvalue.Value (Uint 5),emptyState))"
by (normalization, simp)

```

```

lemma "P (execute (address_monad A5) emptyState)
  = P (Normal (rvalue.Value (Address A5),emptyState))"
by (normalization,simp)

```

```

lemma "is_Normal (execute (do {
  assert_monad (equals_monad (contract_this_monad) (address_monad A1))
}) emptyState)"
by (normalization)

```

```

lemma "is_Normal (execute (do {
  assert_monad (equals_monad (method_sender_monad) (address_monad A1))
}) emptyState)"
by (normalization)

```

```

lemma "is_Normal (execute (do {
  assert_monad (equals_monad (method_value_monad) (sint_monad 250))
}) emptyState)"
by (normalization)

```

```

lemma "is_Normal (execute (do {
  assert_monad (equals_monad (method_timestamp_monad) (sint_monad 100))
}) emptyState)"
by (normalization)

```

```

lemma "is_Normal (execute (do {
  assert_monad (equals_monad (keccak256_keccak256_monad (sint_monad 5)) (sint_monad 5))
}) emptyState)"
by (normalization)

```

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)"
by (normalization)

```

```

lemma "is_Normal (execute (do {
  assert_monad (equals_monad (not_monad false_monad) true_monad)
}) emptyState)"
by (normalization)

```

7.4.2 Equals monad

```

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

```

```

    } emptyState)"
  by (normalization)

```

7.4.3 Less monad

```

lemma "is_Normal (execute (do {
  assert_monad (less_monad (sint_monad 10) (sint_monad 11))
  } emptyState)"
  by (normalization)

```

7.4.4 And monad

```

lemma "is_Normal (execute (do {
  assert_monad (equals_monad (and_monad (true_monad) (false_monad)) (false_monad))
  } emptyState)"
  by (normalization)

```

7.4.5 Or monad

```

lemma "is_Normal (execute (do {
  assert_monad (equals_monad (or_monad (true_monad) (false_monad)) (true_monad))
  } emptyState)"
  by (normalization)

```

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)"
  by (normalization)

```

```

lemma "is_Normal (execute (do {
  assert_monad (equals_monad (plus_monad (sint_monad (2256 - 1)) (sint_monad 6)) (sint_monad 5))
  } emptyState)"
  by (normalization)

```

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)"
  by (normalization)

```

```

lemma "is_Exception (execute (do {
  assert_monad (equals_monad (plus_monad_safe (sint_monad (2256 - 1)) (sint_monad 6)) (sint_monad
5))
  } emptyState)"
  by (normalization)

```

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)"
  by (normalization)

```

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)"
  by (normalization)

```

```

lemma "is_Exception (execute (do {

```

```

    assert_monad (equals_monad (minus_monad_safe (sint_monad 5) (sint_monad 6)) (sint_monad (2256 -
5)))
  }) emptyState)"
  by (normalization)

```

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)"
  by (normalization)

```

```

lemma "is_Normal (execute (do {
  assert_monad (equals_monad (mult_monad (sint_monad (2255)) (sint_monad 2)) (sint_monad 0))
  }) emptyState)"
  by (normalization)

```

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)"
  by (normalization)

```

```

lemma "is_Exception (execute (do {
  assert_monad (equals_monad (mult_monad_safe (sint_monad (2255)) (sint_monad 2)) (sint_monad 0))
  }) emptyState)"
  by (normalization)

```

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)"
  by (normalization)

```

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)" by normalization

```

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)" by normalization

```

```

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)" by normalization
```

```
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)" by normalization
```

```
lemma "is_Exception (execute (do {
  assert_monad (equals_monad (contract_storeLookup (STR ''x'') [sint_monad 0])
(contract_storeLookup (STR ''y'') [sint_monad 0]))
}) state21)" by normalization
```

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)" by normalization
```

```
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)" by normalization
```

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)" by normalization
```

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)" by normalization
```

```

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)" by normalization

```

```

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)" by normalization

```

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)" by normalization

```

```

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)" by normalization

```

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)" by normalization

```

```

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)" by normalization
```

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)" by normalization
```

```
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)" by normalization
```

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)" by normalization
```

```
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)" by normalization
```

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)" by normalization
```

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)" by normalization
```

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)"
by (normalization)
```

```
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)"
by (normalization)
```

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)"
by (normalization)
```

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)"
by (normalization)
```

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)"
by (normalization)
```

```
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)"
  by (normalization)

```

```

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)"
  by (normalization)

```

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)" by normalization

```

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)" by normalization

```

```

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)" by normalization

```

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)" by normalization

```

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)" by normalization

```

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)" by normalization

```

```

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)" by normalization

```

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)"
  by (normalization)

```

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)" by normalization

```

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)" by normalization

```

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)" by normalization

```

```

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)" by normalization

```

```
lemma "state.Memory (snd (normal (execute (do {
  mdecl (DArray (TValue TSint)) (STR ''y''))
  }) emptyState)))=[mdata.Array []]" by normalization
```

```
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)" by normalization
```

```
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)"
  by normalization
```

```
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)"
  by normalization
```

```
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);
```

7 Unit Tests

```
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)"
by normalization
```

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)"
by normalization
```

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)"
by normalization
```

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)"
by normalization
```

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)"
  by normalization

```

context

```

  includes bit_operations_syntax
begin

```

```

lemma "word8_to_char ((char_to_word8 (CHR 0x56)) OR (char_to_word8 (CHR 0x4C))) = CHR 0x5E"
  by (normalization)

```

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)"
  by normalization

```

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)"
  by normalization

```

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)"
  by normalization

```

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'');

```

7 Unit Tests

```
    assert_monad (equals_monad (bytes_cast_monad 4 (contract_stackLookup (STR ''x'') []))
(contract_stackLookup (STR ''y'') []))
    }) emptyState)"
  by normalization
```

```
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)"
  by normalization
```

```
lemma "is_Exception (execute (do {
  bytes_monad 1 []
  }) emptyState)"
  by normalization
```

```
lemma "is_Normal (execute (do {
  bytes_monad 32 (array 32 CHR 0x00)
  }) emptyState)"
  by normalization
```

```
lemma "is_Exception (execute (do {
  bytes_monad 33 (array 33 CHR 0x00)
  }) emptyState)"
  by normalization
```

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  $\equiv$  STR ''balances''"
abbreviation "bal  $\equiv$  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  $\equiv$   $\sum ad \in UNIV. unat (valtype.uint (storage\_data.vt (x (Address ad))))$ "
```

```
context Solidity
begin
```

```
lemma 1:
  fixes bal
  assumes "SUMM bal  $\leq$  Balances s this"
    and "bal (Address msg_sender) = storage_data.Value (Uint y)"
    and "unat y + unat msg_value < 2256"
  shows "( $\sum ad \in UNIV. unat (valtype.uint (storage\_data.vt (if ad = msg\_sender$  then storage_data.Value (Uint (y + msg_value)) else bal (Address ad))))"
```

```

    ≤ Balances s this + unat msg_value"
proof -
  from sum_addr[of _ msg_sender] have "( $\sum ad/ad \in UNIV \wedge ad \neq msg\_sender. unat (valtype.uint (storage\_data.vt (bal (Address ad))))$ ) +
    unat (valtype.uint (storage_data.vt (bal (Address msg_sender)))) + unat msg_value ≤ Balances s this
+ unat msg_value"
  using assms(1) by simp
  moreover have "unat (valtype.uint (storage_data.vt (storage_data.Value (Uint (y + msg_value)))) ≤
unat (valtype.uint (storage_data.vt (bal (Address msg_sender)))) + unat msg_value"
  using assms(2,3) unat_add_lem[where ?'a =256] by simp
  ultimately show ?thesis using sum_addr[of _ msg_sender] by simp
qed

```

lemma 2:

```

  fixes bal
  assumes "SUMM bal ≤ Balances s this"
  and "bal (Address msg_sender) = storage_data.Value (Uint y)"
  shows "( $\sum ad \in UNIV. unat (valtype.uint (storage\_data.vt (if ad = msg\_sender \text{ then } storage\_data.Value (Uint 0) \text{ else } bal (Address ad))))$ )
    ≤ Balances s this + unat msg_value - unat y"

```

proof -

```

  from sum_addr[of _ msg_sender] have "( $\sum ad/ad \in UNIV \wedge ad \neq msg\_sender. unat (valtype.uint (storage\_data.vt (bal (Address ad))))$ ) +
    (unat (valtype.uint (storage_data.vt (bal (Address msg_sender)))) + unat msg_value - unat y) ≤
Balances s this + unat msg_value - unat y"
  using assms(1,2) by simp
  moreover have "unat (valtype.uint (storage_data.vt (storage_data.Value (Uint 0)))) ≤ unat
(valtype.uint (storage_data.vt (bal (Address msg_sender)))) + unat msg_value - unat y"
  using assms(2) unat_add_lem[where ?'a =256] by simp
  ultimately show ?thesis using sum_addr[of _ msg_sender] by simp
qed

```

lemma 3:

```

  fixes bal
  assumes "SUMM bal ≤ Balances s this"
  and "bal (Address msg_sender) = storage_data.Value (Uint y)"
  shows "( $\sum ad \in UNIV. unat (valtype.uint (storage\_data.vt (if ad = msg\_sender \text{ then } storage\_data.Value (Uint 0) \text{ else } bal (Address ad))))$ )
    ≤ Balances s this + unat msg_value"
  using 2[OF assms] by simp

```

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 "( $\sum ad \in UNIV. unat (valtype.uint (storage\_data.vt (if ad = this \text{ then } storage\_data.Value (Uint 0) \text{ else } bal (Address ad))))$ )
    ≤ Balances s this + unat msg_value"
  using 3[OF assms(1-2)] assms(3) by simp

```

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)"
  using assms by auto

```

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 -
  show "∧ call.
    (∧ x h r. effect (call x) h r ⇒ vcond x h r) ⇒
      effect (constructor call) s r ⇒ post s r sum_bal (K True) (K (K (K True)))"
    unfolding constructor_def
    apply (erule post_exc_true, erule_tac post_wp)
    unfolding inv_state_def
  by (vcg wprules: sum_balI | auto)+

  show "∧ call.
    (∧ x h r. effect (call x) h r ⇒ vcond x h r) ⇒
      effect (deposit call) s r ⇒ inv_state sum_bal s ⇒ post s r sum_bal (K True) deposit_post"
    unfolding deposit_def
    apply (erule post_exc_true, erule_tac post_wp)
    unfolding inv_state_def deposit_post_def
    apply vcg
    apply (auto simp add:wpsimps)
    apply (rule bal_msg_sender, assumption)
    apply vcg
    apply (auto simp add: wpsimps intro!: sum_balI 1)
    apply vcg
    apply (auto simp add:wpsimps)
    apply (rule bal_msg_sender, assumption)
    by vcg

  show "∧ call.
    (∧ x h r. effect (call x) h r ⇒ vcond x h r) ⇒
      effect (withdraw call) s r ⇒ inv_state sum_bal s ⇒ post s r sum_bal (K True) (K (K (K
True)))"
    unfolding withdraw_def
    apply (erule post_exc_true, erule_tac post_wp)
    unfolding inv_state_def
    apply (case_tac "msg_sender = this")
    apply (vcg)
    apply (auto simp add:wpsimps)
    apply (rule_tac bal_msg_sender, assumption)
    apply (vcg)
    apply (rule_tac s = msg_sender in subst, assumption)
    apply (vcg)
    apply (rule subst, assumption)
    apply (vcg)
    apply (rule subst, assumption)
    apply (vcg)
    apply (auto simp add:wpsimps)
    apply (vcg)
    apply (auto simp add:wpsimps)

```

```

    apply (vcg)
    apply (rule_tac mp = "mp(Address msg_sender := storage_data.Value (Uint 0))" in sum_balI)
    apply vcg
    apply (auto simp add:wpsimps 4)
    apply vcg
    apply (rule_tac mp = mpa in sum_balI)
    apply vcg
    apply (auto)
    apply vcg
    apply (auto simp add:wpsimps)
    apply (rule bal_msg_sender, assumption)
    apply vcg
    apply (rule_tac mp = "mp(Address msg_sender := storage_data.Value (Uint 0))" in sum_balI)
    apply (simp add:2 wpsimps)
    apply (simp add:wpsimps)
    apply vcg
  done
qed

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 \equiv STR ''balances''"

abbreviation "bal \equiv 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 $x \equiv \sum ad \in UNIV. \text{unat} (\text{valtype.unt} (\text{storage_data.vt} (x (\text{Address } ad))))$ "

```

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.unt (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 -

```

    from sum_addr[of _ msg_sender] have "(∑ ad / ad ∈ UNIV ∧ ad ≠ msg_sender. unat (valtype.unt
(storage_data.vt (bal (Address ad)))))) +
    unat (valtype.unt (storage_data.vt (bal (Address msg_sender)))) + unat msg_value ≤ Balances s this
+ unat msg_value"
    using assms(1) by simp
    moreover have "unat (valtype.unt (storage_data.vt (storage_data.Value (Uint (y + msg_value)))))) ≤
unat (valtype.unt (storage_data.vt (bal (Address msg_sender)))) + unat msg_value"
    using assms(2,3) unat_add_lem[where ?'a =256] by simp
    ultimately show ?thesis using sum_addr[of _ msg_sender] by simp
qed

```

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 -

```

    from sum_addr[of _ msg_sender] have
    "(∑ ad / ad ∈ UNIV ∧ ad ≠ msg_sender. unat (valtype.unt (storage_data.vt (bal (Address ad)))))) +
    (unat (valtype.unt (storage_data.vt (bal (Address msg_sender)))) - unat y)
    ≤ Balances s this - unat y"
    using assms(1,2) by simp
    moreover have "unat (valtype.unt (storage_data.vt (storage_data.Value (Uint 0)))) ≤ unat
(valtype.unt (storage_data.vt (bal (Address msg_sender)))) + unat msg_value - unat y"
    using assms(2) unat_add_lem[where ?'a =256] by simp
    ultimately show ?thesis using assms sum_addr[of _ msg_sender] by auto
qed

```

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"
    using 21[OF assms] by simp

```

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)"
using assms by auto

```

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 -
```

```

show "∧call.
  (∧x h r. effect (call x) h r ⇒ vcond x h r) ⇒
  effect (constructor call) s r ⇒ post s r sum_bal (K True) (K (K (K True)))"
unfolding constructor_def
apply (erule post_exc_true, erule_tac post_wp)
unfolding inv_state_def
by (vcg wprules: sum_balI | auto)+

```

```

show "∧call. effect (reset call) s r ⇒
  (∧x h r. effect (call x) h r ⇒ vcond x h r) ⇒ post s r (K True) (K True) reset_post"
unfolding reset_def
apply (erule post_exc_true, erule_tac post_wp)
unfolding inv_state_def
apply vcg
by auto

```

```

show "∧call.
  (∧x h r. effect (call x) h r ⇒ vcond x h r) ⇒
  effect (deposit call) s r ⇒ inv_state sum_bal s ⇒ post s r sum_bal (K True) (K (K (K
True)))"

```

```

unfolding deposit_def
apply (erule post_exc_true, erule_tac post_wp)
unfolding inv_state_def
apply vcg
apply (auto simp add: wpsimps)
apply (rule bal_msg_sender, assumption)
apply vcg
apply (auto simp add: wpsimps intro!: sum_balI 1)
apply vcg
apply (auto simp add: wpsimps)
apply (rule bal_msg_sender, assumption)
by vcg

```

```

show " $\wedge$ call.
  ( $\wedge$ x h r. effect (call x) h r  $\implies$  vcond x h r)  $\implies$ 
  effect (withdraw call) s r  $\implies$  inv_state sum_bal s  $\implies$  post s r sum_bal (K True) (K (K (K
True)))"
  unfolding withdraw_def
  apply (erule post_exc_true, erule_tac post_wp)
  unfolding inv_state_def icall_def
  apply (case_tac "msg_sender = this")
  apply (vcg)
  apply (rule_tac s = msg_sender in subst, assumption)
  apply (vcg)

  apply (subgoal_tac " $\wedge$ x h r. effect (call x) h r  $\implies$  vcond x h r")
    apply (rule_tac c=call and x=Reset_m and P'=reset_post in wp_post)
  using vcond(3) apply simp apply blast

  apply (vcg)
  apply (rule_tac s=msg_sender in subst, assumption)
  apply (vcg)
  apply (auto simp add:wpsimps)
  apply (vcg)
  apply (auto simp add:wpsimps)
  apply (rule bal_msg_sender, assumption)
  apply (vcg)
  apply (rule_tac mp = mp' in sum_balI)
  apply (auto simp add:wpsimps intro: 22)
  apply (vcg)
  apply (rule_tac mp = mpa in sum_balI)
  apply (vcg)

  apply (subgoal_tac " $\wedge$ x h r. effect (call x) h r  $\implies$  vcond x h r")
    apply (rule_tac c=call and x=Reset_m and P'=reset_post in wp_post)
  using vcond(3) apply simp apply blast

  apply vcg
  apply (auto simp add:wpsimps)
  apply vcg
  apply (auto simp add:wpsimps)
  apply (rule bal_msg_sender, assumption)
  apply vcg
  apply (rule_tac mp = mp' in sum_balI)
  apply (auto simp add:wpsimps intro: 21)
  apply vcg
  apply (rule_tac mp = mpa in sum_balI)
  apply vcg
done
qed

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.com>.

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 <shint> 1
    }",
```

```
cfunction placeBet external payable
```

```
  param guess: "SType.TValue TSint"
  where
    "do {
      inState (valtype.Uint 1);
      <assert> (<¬> (<sender> <=> operator ~s []));
      <assert> (<value> <<> (pot ~s []) <V> <value> <=> (pot ~s []));
      state [] ::=s <shint> 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 ~ []) <%> <shint> 2) <=> (<shint> 0) THEN <shint> HEADS ELSE <shint>
TAILS;
      IF (secret ~ []) <=> (guess ~s []) THEN
        do {
          pot [] ::=s ((pot ~s []) <-> (bet ~s []));
          bet [] ::=s <shint> 0;
          <transfer> (player ~s []) ((bet ~s []) <*> (<shint> 2))
        }
      ELSE
        do {
          pot [] ::=s pot ~s [] <+> bet ~s [];
          bet [] ::=s <shint> 0
        };
      state [] ::=s <shint> 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"
using assms by simp

```

```

lemma kdequals_true_dest[wpdrules]:
  assumes "kdequals a b = Some (rvalue.Value (Bool True))"
  shows "a = b"
using assms
apply (cases a;simp add: kdequals_def)
apply (cases b;simp add: kdequals_def)
by (case_tac x4;case_tac x4a;simp add: kdequals_def split: if_split_asm)

```

```

lemma kdequals_false_dest[wpdrules]:
  assumes "kdequals a b = Some (rvalue.Value (Bool False))"
  shows "a ≠ b"
using assms
apply (cases a;simp add: kdequals_def)
apply (cases b;simp add: kdequals_def)
by (case_tac x4;case_tac x4a;simp add: kdequals_def split: if_split_asm)

```

```

lemma kdnnot_dest[wpdrules]:
  assumes "kdnnot ya = Some (rvalue.Value (Bool True))"
  shows "ya=rvalue.Value (Bool False)"
using assms apply (cases ya;simp add:assms kdnnot_def)
by (case_tac x4;simp add:assms vtnot_def)

```

```

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)"
using assms
apply (cases ye;simp add: vtor_def)

```

```

apply (cases yg;simp add: vtor_def)
by (case_tac x4;case_tac x4a;simp)

lemma kdless_dest[wpdrules]:
  assumes "kdless (rvalue.Value (Uint x)) (rvalue.Value (Uint y)) = Some (rvalue.Value (Bool True))"
  shows "x < y"
  using assms
  apply (cases x;simp add: kdless_def)
  apply (cases y;simp add: kdless_def)
  by (case_tac n;case_tac na;simp add: vtless_def)

lemma kdequals_bool_sint[wpsimps]: "kdequals (rvalue.Value (Bool x)) (rvalue.Value (Uint y)) = None"
  unfolding kdequals_def by simp

lemma unat_add_imp:
  assumes "(unat (x::256 word) + unat y < 115792089237316195423570985008687907853269984665640564039457584007913129)"
  shows "(unat (x + y) = unat x + unat y)" using assms Word.unat_add_lem[where ?'a=256, of x y] by
  simp

lemma unat_mult_imp:
  assumes "(unat (x::256 word) * unat y < 115792089237316195423570985008687907853269984665640564039457584007913129)"
  shows "(unat (x * y) = unat x * unat y)" using assms Word.unat_mult_lem[where ?'a=256, of x y] by
  simp

lemma unat_sub_imp:
  assumes "(y::256 word) ≤ (x::256 word)"
  shows "(unat (x - y) = unat x - unat y)" using assms Word.unat_sub[where ?'a=256, of y x] by simp

lemma no_plus_overflow_leq:
  assumes a:"(x ::256 word) ≤ y"
  and b: "unat y + unat z < 2 ^ 256"
  shows "x ≤ y + z" using no_plus_overflow_unat_size[of z y] le_no_overflow[OF a, of z]
proof -
  have "size z = 256" using word_size[of z] by simp
  with b have "z ≤ z + y" using no_plus_overflow_unat_size[of z y] by simp
  then have "x ≤ z + y" using le_no_overflow[OF a, of z] by simp
  then show ?thesis by (simp add: add.commute)
qed

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"
  using assms unat_sub[OF *] by simp

lemma 222: "∧pot. unat pot ≤ Balances s this ⇒
  x ≤ pot ⇒
  unat (pot - x) ≤ Balances s this + unat msg_value"
  using 111 by fastforce

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"
  using assms by auto

lemma notwp[wperules]: "¬ wp m P E s ⇒ wp m P E s ⇒ R" using notE by simp

```

```

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 (K True)" "K (K (K True))" and
  "removeFromPot" "K (K True)" "K (K (K (K True)))"
  for "casino"
proof -
  show "∧call. (∧x h r. effect (call x) h r ⇒ vcond x h r)
    ⇒ effect (constructor call) s r
    ⇒ post s r pot_balance (K True) (K (K (K True)))"
    unfolding constructor_def
    apply (erule post_exc_true, erule_tac post_wp)
    unfolding inv_state_def
  by (vcg wprules: pot_balanceI | auto)+

  show "∧call hashNum. (∧x h r. effect (call x) h r ⇒ vcond x h r)
    ⇒ effect (createGame call hashNum) s r
    ⇒ inv_state pot_balance s
    ⇒ post s r pot_balance (K True) (createGame_post hashNum)"
    unfolding createGame_def
    apply (erule post_exc_true, erule_tac post_wp)
    unfolding inv_state_def
  by (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps>)+

  show "∧call guess. (∧x h r. effect (call x) h r ⇒ vcond x h r)
    ⇒ effect (placeBet call guess) s r
    ⇒ inv_state pot_balance s
    ⇒ post s r pot_balance (K True) (placeBet_post guess)"
    unfolding placeBet_def
    apply (erule post_exc_true, erule_tac post_wp)
    unfolding inv_state_def
    apply (vcg | solve<auto simp add:wpsimps dest:sym>)+
    apply (auto dest!: vt_or_dest)[1]
    apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps dest:sym>)+
  done

  show "∧call secretNumber. (∧x h r. effect (call x) h r ⇒ vcond x h r)
    ⇒ effect (decideBet call secretNumber) s r
    ⇒ inv_state pot_balance s
    ⇒ post s r pot_balance (K True) (decideBet_post secretNumber)"
    unfolding decideBet_def
    apply (erule post_exc_true, erule_tac post_wp)
    unfolding inv_state_def
    apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps>)+
    apply (auto simp add:wpsimps)[1]
    apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps>)+
    apply (auto simp add:wpsimps unat_sub_imp)[1]

```

```

apply (case_tac "this = ada")
apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps>)+
apply (auto simp add:wpsimps dest:sym)[1]
apply (case_tac "this = ada")
apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps dest:sym2>)+
apply (auto simp add:wpsimps)[1]
apply (metis trans_le_add1 unat_add_imp)
apply (auto simp add:wpsimps dest:sym)[1]
apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps dest:sym>)+
apply (auto simp add:wpsimps)[1]
apply (case_tac "this = ada")
apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps dest:sym>)+
apply (auto simp add:wpsimps dest:sym)[1]
apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps dest:sym>)+
apply (auto simp add:wpsimps unat_sub_imp dest:sym)[9]
apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps dest:sym2>)+
apply (auto simp add:wpsimps unat_sub_imp dest:sym)[9]
apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps dest:sym2>)+
apply (auto simp add:wpsimps)[1]
apply (case_tac "this = ada")
apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps dest:sym2>)+
apply (auto simp add:wpsimps)[1]
apply (metis trans_le_add1 unat_add_imp)
apply (metis trans_le_add1 unat_add_imp)
apply (auto simp add:wpsimps dest:sym)
apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps>)+
done

show "\call.
  (\x h r. effect (call x) h r  $\implies$  vcond x h r)  $\implies$ 
  effect (addToPot call) s r  $\implies$ 
  inv_state pot_balance s  $\implies$  post s r pot_balance (K True) (K (K (K True)))"
unfolding addToPot_def
apply (erule post_exc_true, erule_tac post_wp)
unfolding inv_state_def
apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps dest:unat_add_imp sym>)+
apply (auto simp add:wpsimps no_plus_overflow_leq unat_sub_imp unat_mult_imp unat_add_imp dest:sym)
apply (vcg wprules: pot_balanceI | solve<auto simp add:wpsimps dest:unat_add_imp sym>)+
done

show "\call amount.
  (\x h r. effect (call x) h r  $\implies$  vcond x h r)  $\implies$ 
  effect (removeFromPot call amount) s r  $\implies$ 
  inv_state pot_balance s  $\implies$ 
  post s r pot_balance (K True) (K (K (K (K True))) amount)"
unfolding removeFromPot_def
apply (erule post_exc_true, erule_tac post_wp)
apply (cases "msg_sender=this")
unfolding inv_state_def
apply (vcg | solve<auto simp add:wpsimps dest:unat_add_imp sym>)+
apply (drule_tac s = this in sym)
apply (auto simp add:wpsimps dest:unat_add_imp sym)
apply (vcg | solve<auto simp add:wpsimps dest:unat_add_imp sym>)+
apply (auto simp add:wpsimps dest:unat_add_imp sym)
apply (vcg | solve<auto simp add:wpsimps dest:unat_add_imp sym>)+
apply (auto simp add:wpsimps dest:unat_add_imp sym)
apply (vcg | solve<auto simp add:wpsimps dest:unat_add_imp sym>)+
apply (rule_tac pot_balanceI)
apply (auto simp add:wpsimps intro: 111 222 dest:unat_add_imp sym)
apply (vcg | solve<auto simp add:wpsimps dest:unat_add_imp sym>)+
apply (rule_tac pot_balanceI)
apply (auto simp add:wpsimps dest:unat_add_imp sym)
apply (vcg | solve<auto simp add:wpsimps dest:unat_add_imp sym>)+
apply (auto simp add:wpsimps dest:unat_add_imp sym)

```

```

  apply (vcg | solve<auto simp add:wpsimps dest:unat_add_imp sym>)+
  apply (auto simp add:wpsimps dest:unat_add_imp sym)
  apply (vcg | solve<auto simp add:wpsimps dest:unat_add_imp sym>)+
  apply (rule_tac pot_balanceI)
  apply (auto simp add:wpsimps intro: 111 dest:unat_add_imp sym)
  apply (vcg | solve<auto simp add:wpsimps dest:unat_add_imp sym>)+
  apply (rule_tac pot_balanceI)
  apply (auto simp add:wpsimps dest:unat_add_imp sym)
  apply (vcg | solve<auto simp add:wpsimps dest:unat_add_imp sym>)+
done
qed

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"
  using assms unfolding kdequals_def by simp

```

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)) ≡ $\sum ad/\exists v. x$ (Address ad) = storage_data.Array v ∧ valtype.bool (storage_data.vt (v ! 1)). (THE y. $\exists v. x$ (Address ad) = storage_data.Array v ∧ y=unat (valtype.uint (storage_data.vt (v ! 0))))"

abbreviation "SUMM2 (x::'a valtype storage_data list) ≡ $\sum i<\text{length } x. (\text{THE } y. \exists p. x ! i = \text{storage_data.Array } p \wedge y=\text{unat } (\text{valtype.uint } (\text{storage_data.vt } (p ! 1))))$ "

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)]"
  using assms by fastforce

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)]"
  using assms by blast

lemma kdequals_true[wpdrules]:
  assumes "kdequals (rvalue.Value (Address x)) (rvalue.Value (Address y)) = Some (rvalue.Value (Bool
    True))"
  shows "x = y"
  using assms unfolding kdequals_def by simp

lemma kdequals_false:
  assumes "kdequals (rvalue.Value (Address x)) (rvalue.Value (Address y)) = Some (rvalue.Value (Bool
    False))"
  shows "x ≠ y"
  using assms unfolding kdequals_def by simp

lemma kdnnot[wpdrules]:
  assumes "kdnnot (rvalue.Value (Bool t)) = Some (rvalue.Value (Bool True))"
  shows "¬ t"
  using assms unfolding kdnnot_def vtnot_def by simp

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)"
  unfolding inv'_def
proof ((rule allI | rule impI)+, erule conjE)
  fix xa y
  define sum1 where "sum1 = (λ(y::'a valtype storage_data list) i . THE ya. ∃p. y ! i =
    storage_data.Array p ∧ ya = unat (valtype.uint (storage_data.vt (p ! 1))))"
  define sum2 where "sum2 = (λ(xa::'a valtype ⇒ 'a valtype storage_data) ad. THE y. ∃v. xa (Address
    ad) = storage_data.Array v ∧ y = unat (valtype.uint (storage_data.vt (v ! 0))))"
  assume *: "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 STR ''voters'' = Map xa"
  and **: "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 STR ''proposals'' = storage_data.Array y"
  show "( $\sum i < \text{length } y. \text{sum1 } y \ i$ )  $\leq$  ( $\sum ad \mid \exists v. xa \text{ (Address } ad) = \text{storage\_data.Array } v \wedge$ 
  valtype.bool (storage_data.vt (v ! 1)). sum2 xa ad)"
  proof -
    from assms have "( $\sum i < \text{length } \text{proposals}. \text{sum1 } \text{proposals } i$ )  $\leq$  ( $\sum ad \mid \exists v. \text{voters (Address } ad)$ 
  = storage_data.Array v  $\wedge$  valtype.bool (storage_data.vt (v ! 1)). sum2 voters ad)" unfolding inv'_def
  sum1_def sum2_def by simp
    moreover have "( $\sum i < \text{length } \text{proposals}. \text{sum1 } \text{proposals } i$ ) = ( $\sum i < \text{length } y. \text{sum1 } y \ i$ )" using
  assms ** unfolding sum1_def by (simp add:wpsimps)
    moreover have "( $\sum ad \mid \exists v. \text{voters (Address } ad) = \text{storage\_data.Array } v \wedge \text{valtype.bool}$ 
  (storage_data.vt (v ! 1)). sum2 voters ad) = ( $\sum ad \mid \exists v. xa \text{ (Address } ad) = \text{storage\_data.Array } v \wedge$ 
  valtype.bool (storage_data.vt (v ! 1)). sum2 xa ad)"
    proof -
      from assms have " $\nexists v. \text{voters (Address } x) = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt}$ 
  (v ! 1))" by simp
      then have "( $\sum ad \mid \exists v. \text{voters (Address } ad) = \text{storage\_data.Array } v \wedge \text{valtype.bool}$ 
  (storage_data.vt (v ! 1)). sum2 voters ad) =
      ( $\sum ad \mid (\exists v. \text{voters (Address } ad) = \text{storage\_data.Array } v \wedge \text{valtype.bool}$ 
  (storage_data.vt (v ! 1)))  $\wedge ad \neq x. \text{sum2 voters ad}$ )" using sum_addr3[of x "{ad.  $\exists v. \text{voters (Address}$ 
  ad) = storage_data.Array v  $\wedge$  valtype.bool (storage_data.vt (v ! 1))}]" "sum2 voters"] by simp
      moreover have " $\neg (\exists v. xa \text{ (Address } x) = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v}$ 
  ! 1)))" using * by (auto simp add:wpsimps)
      then have "( $\sum ad \mid \exists v. xa \text{ (Address } ad) = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt}$ 
  (v ! 1)). sum2 xa ad) =
      ( $\sum ad \mid (\exists v. xa \text{ (Address } ad) = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt}$ 
  (v ! 1)))  $\wedge ad \neq x. \text{sum2 xa ad}$ )" using sum_addr3[of x "{ad.  $\exists v. xa \text{ (Address } ad) = \text{storage\_data.Array}$ 
  v  $\wedge$  valtype.bool (storage_data.vt (v ! 1))}]" "sum2 xa"] by simp
      moreover have "( $\sum ad \mid (\exists v. \text{voters (Address } ad) = \text{storage\_data.Array } v \wedge \text{valtype.bool}$ 
  (storage_data.vt (v ! 1)))  $\wedge ad \neq x. \text{sum2 voters ad}$ )
      = ( $\sum ad \mid (\exists v. xa \text{ (Address } ad) = \text{storage\_data.Array } v \wedge \text{valtype.bool}$ 
  (storage_data.vt (v ! 1)))  $\wedge ad \neq x. \text{sum2 xa ad}$ )"
      proof (rule sum.mono_neutral_cong_right)
        show "finite {ad. ( $\exists v. \text{voters (Address } ad) = \text{storage\_data.Array } v \wedge \text{valtype.bool}$ 
  (storage_data.vt (v ! 1)))  $\wedge ad \neq x$ }" using finite finite_subset subset_UNIV by blast
        show "{ad. ( $\exists v. xa \text{ (Address } ad) = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v}$ 
  ! 1)))  $\wedge ad \neq x$ }
           $\subseteq$  {ad. ( $\exists v. \text{voters (Address } ad) = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v}$ 
  ! 1)))  $\wedge ad \neq x$ }" using * by (auto simp add:wpsimps)
        show " $\forall i \in \{ad. (\exists v. \text{voters (Address } ad) = \text{storage\_data.Array } v \wedge \text{valtype.bool}$ 
  (storage_data.vt (v ! 1)))  $\wedge ad \neq x\}$  -
          {ad. ( $\exists v. xa \text{ (Address } ad) = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v}$ 
  ! 1)))  $\wedge ad \neq x$ }.
          sum2 voters i = 0" using * by (auto simp add:wpsimps)
        show " $\bigwedge xb. xb \in \{ad. (\exists v. xa \text{ (Address } ad) = \text{storage\_data.Array } v \wedge \text{valtype.bool}$ 
  (storage_data.vt (v ! 1)))  $\wedge ad \neq x\} \implies \text{sum2 voters } xb = \text{sum2 xa } xb$ " using * unfolding sum2_def
  by (auto simp add:wpsimps)
      qed
      ultimately show ?thesis by simp
    qed
    ultimately show ?thesis by simp
  qed
qed

```

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  $\neq$  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"
unfolding inv'_def
proof ((rule allI | rule impI)+, erule conjE)
fix x y
define sum1 where "sum1 = (λ(y::'a valtype storage_data list) i . THE ya. ∃p. y ! i =
storage_data.Array p ∧ ya = unat (valtype.uint (storage_data.vt (p ! 1))))"
define sum2 where "sum2 = (λ(xa::'a valtype ⇒ 'a valtype storage_data) ad. THE y. ∃v. xa (Address
ad) = storage_data.Array v ∧ y = unat (valtype.uint (storage_data.vt (v ! 0))))"
assume *: "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 STR ''voters'' = Map x"
and **: " 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 STR ''proposals'' = storage_data.Array y"
show "(∑ i < length y. sum1 y i) ≤ (∑ ad | ∃v. x (Address ad) = storage_data.Array v ∧ valtype.bool
(storage_data.vt (v ! 1)). sum2 x ad)"

```

```

proof -
  from assms have " $(\sum i < \text{length proposals. sum1 proposals } i) \leq (\sum \text{ad} \mid \exists v. \text{voters (Address ad)})$ " unfolding inv'_def
sum1_def sum2_def by simp
  moreover have " $(\sum i < \text{length } y. \text{sum1 } y \ i) = (\sum i < \text{length proposals. sum1 proposals } i)$ "
    using * ** assms by (simp add:wpsimps)
  moreover have " $(\sum \text{ad} \mid \exists v. \text{voters (Address ad)}) = \text{storage\_data.Array } v \wedge \text{valtype.bool}$ 
(storage_data.vt (v ! 1)). sum2 voters ad)  $\leq (\sum \text{ad} \mid \exists v. x \text{ (Address ad)} = \text{storage\_data.Array } v \wedge$ 
valtype.bool (storage_data.vt (v ! 1)). sum2 x ad)"
```

proof -

```

  have " $(\sum \text{ad} \mid (\exists v. x \text{ (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v !$ 
1))  $\wedge \text{ad} \neq \text{xa}) \wedge \text{ad} \neq \text{msg\_sender. sum2 } x \text{ ad}) = (\sum \text{ad} \mid (\exists v. \text{voters (Address ad)} = \text{storage\_data.Array}$ 
v  $\wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge \text{ad} \neq \text{xa}) \wedge \text{ad} \neq \text{msg\_sender. sum2 voters ad})"$ "
  proof (rule sum.mono_neutral_cong_right)
    show "finite {ad. ( $\exists v. x \text{ (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{xa}) \wedge \text{ad} \neq \text{msg\_sender}$ }>" using finite finite_subset subset_UNIV by blast
    show "{ad. ( $\exists v. \text{voters (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{xa}) \wedge \text{ad} \neq \text{msg\_sender}$ }
 $\subseteq$  {ad. ( $\exists v. x \text{ (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{xa}) \wedge \text{ad} \neq \text{msg\_sender}$ }>" using * by (auto simp add:wpsimps)
    show " $\forall i \in \{\text{ad. } (\exists v. x \text{ (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{xa}) \wedge \text{ad} \neq \text{msg\_sender}\}$  -
{ad. ( $\exists v. \text{voters (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{xa}) \wedge \text{ad} \neq \text{msg\_sender}$ }."
    sum2 x i = 0" using * by (auto simp add:wpsimps)
    show " $\bigwedge x'. x' \in \{\text{ad. } (\exists v. \text{voters (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{xa}) \wedge \text{ad} \neq \text{msg\_sender}\} \implies \text{sum2 } x \ x' = \text{sum2 voters } x'$ " using *
unfolding sum2_def by (auto simp add:wpsimps)
  qed
  moreover have " $(\sum \text{ad} \mid (\exists v. x \text{ (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{xa}) \wedge \text{ad} \neq \text{msg\_sender. sum2 } x \text{ ad})$ 
=  $(\sum \text{ad} \mid (\exists v. x \text{ (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{msg\_sender. sum2 } x \text{ ad})$ " using * ** by (auto simp add:wpsimps)
  moreover have " $(\sum \text{ad} \mid (\exists v. \text{voters (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{xa}) \wedge \text{ad} \neq \text{msg\_sender. sum2 voters ad})$ 
=  $(\sum \text{ad} \mid (\exists v. \text{voters (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{msg\_sender. sum2 voters ad})$ " using assms(7) apply (simp add:wpsimps)
unfolding sum2_def
  by (metis (no_types, lifting) nth_Cons_0 nth_Cons_Suc storage_data.inject(2)
storage_data.sel(1) valtype.sel(1))
  ultimately have " $(\sum \text{ad} \mid (\exists v. x \text{ (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{msg\_sender. sum2 } x \text{ ad})$ 
=  $(\sum \text{ad} \mid (\exists v. \text{voters (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{msg\_sender. sum2 voters ad})$ " by simp
  moreover have "sum2 voters msg_sender = sum2 x msg_sender" using * assms unfolding sum2_def by
(auto simp add:wpsimps)
  moreover have " $(\sum \text{ad} \mid \exists v. \text{voters (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
sum2 voters ad)
=  $(\sum \text{ad} \mid (\exists v. \text{voters (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{msg\_sender. sum2 voters ad})$ "
  proof -
    have " $\neg (\exists v. \text{voters (Address msg\_sender)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))})$ " using assms by (auto simp add:wpsimps)
    then have " $(\sum \text{ad} \mid \exists v. \text{voters (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
sum2 voters ad) =
 $(\sum \text{ad} \mid (\exists v. \text{voters (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
ad  $\neq \text{msg\_sender. sum2 voters ad})$ " using sum_addr3[of msg_sender "{ad.  $\exists v. \text{voters (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))}$ }]" "sum2
voters"] by simp
    then show ?thesis by simp
  qed
  moreover have " $(\sum \text{ad} \mid \exists v. x \text{ (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool (storage\_data.vt (v ! 1))} \wedge$ 
sum2 x ad)
=  $(\sum \text{ad} \mid (\exists v. x \text{ (Address ad)} = \text{storage\_data.Array } v \wedge \text{valtype.bool$ 
```

```

(storage_data.vt (v ! 1))) ∧ ad ≠ msg_sender. sum2 x ad) + sum2 x msg_sender"
  proof -
    have "(∃v. x (Address msg_sender) = storage_data.Array v ∧ valtype.bool (storage_data.vt (v ! 1)))" using assms * by (auto simp add:wpsimps)
    then have "(∑ ad | ∃v. x (Address ad) = storage_data.Array v ∧ valtype.bool (storage_data.vt (v ! 1)). sum2 x ad) =
      (∑ ad | (∃v. x (Address ad) = storage_data.Array v ∧ valtype.bool (storage_data.vt (v ! 1))) ∧ ad ≠ msg_sender. sum2 x ad) + sum2 x msg_sender" using sum_addr2[of msg_sender "{ad. ∃v. x (Address ad) = storage_data.Array v ∧ valtype.bool (storage_data.vt (v ! 1))}" "sum2 x"] by simp
    then show ?thesis by simp
  qed
  ultimately show ?thesis by simp
qed
ultimately show ?thesis by simp
qed

```

```

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"
  apply vcg using assms by auto

```

```

lemma slookup_some[wpdrules]:

```

```

  assumes "proposals $ x ≧≧ slookup [Uint 1] = Some yp"
  shows "x < length proposals ∧ slookup [Uint 1] (proposals ! x) = Some yp"
  using assms unfolding nth_safe_def by (auto split:if_split_asm simp add:wpsimps)

```

```

lemma kdplus_safe_storage_none[wpsimps]:

```

```

  shows "kdplus_safe x (rvalue.Storage p) = None"
  unfolding kdplus_safe_def by simp

```

```

lemma kdplus_safe_storage_some_false[wpdrules]:

```

```

  assumes "kdplus_safe x (rvalue.Storage p) = Some a"
  shows "False"
  using assms by (simp add:wpsimps)

```

```

lemma kdequals2[wpdrules]:

```

```

  assumes "kdequals x y = Some (rvalue.Value (Bool True))"
  shows "x = y"
  using assms
  apply (cases x;simp add: kdequals_def)
  apply (cases y;simp add: kdequals_def)
  by (case_tac x4;case_tac x4a;simp add: kdequals_def split: if_split_asm)

```

```

lemma kdequals3[wpdrules]:

```

```

  assumes "kdequals x y = Some (rvalue.Value (Bool False))"
  shows "¬ x = y"
  using assms
  apply (cases x;simp add: kdequals_def)
  apply (cases y;simp add: kdequals_def)
  by (case_tac x4;case_tac x4a;simp add: kdequals_def split: if_split_asm)

```

```

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"
  unfolding less_monad_def apply (rule wp_lift_op_monad) using assms .

```

```

lemma kdnot_true[wpdrules]:

```

```

  assumes "kdnot y = Some (rvalue.Value (Bool True))"
  shows "y=rvalue.Value (Bool False)"
  using assms apply (cases y;simp add:assms kdnot_def)

```

```

by (case_tac x4; simp add: assms vtnot_def)

lemma list_update_safe_simps1[wpsimps]:
  assumes "i < length xs"
  shows "list_update_safe xs i a = Some (list_update xs i a)"
  unfolding list_update_safe_def using assms by simp
end

lemma notwp[wperules]: "¬ wp m P E s ⇒ wp m P E s ⇒ R" using notE by simp
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 -
  show "∧ call proposalNames. (∧ x h r. effect (call x) h r ⇒ vcond x h r) ⇒
    effect (constructor call proposalNames) s r ⇒ post s r sum_votes (K True) (K (K (K (K True))))
  proposalNames)"
  unfolding constructor_def
  apply (erule post_exc_true, erule_tac post_wp)
  unfolding inv_state_def
  by (vcg wprules: sum_votesI | auto)+
next
  show "∧ call voter. (∧ x h r. effect (call x) h r ⇒ vcond x h r) ⇒
    effect (giveRightToVote call voter) s r ⇒ inv_state sum_votes s ⇒ post s r sum_votes (K
  True) (K (K (K (K True)))) voter)"
  unfolding giveRightToVote_def
  apply (erule post_exc_true, erule_tac post_wp)
  unfolding inv_state_def
  apply (vcg | solve<auto simp add:wpsimps>)+
  apply (auto simp add:wpsimps intro!: wpthrow)
  apply (rule_tac voters=mp and x="Address voter" in obtain_voters, assumption)
  apply (wp | auto simp add:wpsimps)+
  apply (rule_tac mp="mp(Address voter := storage_data.Array [storage_data.Value (Uint 1),
  storage_data.Value (Bool False), storage_data.Value (Address d), storage_data.Value (Uint v)])" in
  sum_votesI)
  apply (wp | auto simp add:wpsimps dest:sym)+
  apply (rule inv_0; assumption)
  apply (wp | auto simp add:wpsimps)+
done

show "∧ call to. (∧ x h r. effect (call x) h r ⇒ vcond x h r) ⇒
  effect (delegate call to) s r ⇒ inv_state sum_votes s ⇒ post s r sum_votes (K True) (K (K (K
  (K True)))) to)"
  unfolding delegate_def
  apply (erule post_exc_true, erule_tac post_wp)
  unfolding inv_state_def
  apply wp
  apply wp
  apply wp
  apply (rule_tac voters=mp and x="Address msg_sender" in obtain_voters, blast)
  apply (vcg | solve<auto simp add:wpsimps>)+

```

```

  apply (rule_tac iv ="λs'. state.Storage s' this STR ''voters'' = state.Storage s this STR
''voters'' ^
                                state.Storage s' this STR ''proposals'' = state.Storage s this STR
''proposals'' ^
                                state.Storage s' this STR ''chairperson'' = state.Storage s this STR
''chairperson'' ^
                                state.Stack s' $$ (STR ''sender'') = Some (kdata.Storage (Some
(Location=STR ''voters'', Offset= [Address msg_sender]))) ^
                                (∃x. (Stack s' $$ STR ''to'' = Some (kdata.Value (Address x)) ^ x ≠
msg_sender))" in wwhile)
  apply (vcg | solve<auto simp add:wpsimps>)+
  apply (safe)[1]
  apply (auto simp add:wpsimps intro!: wpthrow)
  apply (vcg | solve<auto simp add:wpsimps>)+
  apply (auto simp add:wpsimps intro!: wpthrow)
  apply (auto simp add:wpsimps dest: sym)[1]
  apply (vcg | solve<auto simp add:wpsimps>)+
  apply (auto simp add:wpsimps intro!: wpthrow)
  apply (rule_tac voters=mp and x="Address x" in obtain_voters, assumption)
  apply (auto simp add:wpsimps)[1]
  apply (rule_tac voters=mp and x="Address x" in obtain_voters, assumption)
  apply (auto simp add:wpsimps)[1]
  apply (rule_tac voters=mp and x="Address x" in obtain_voters, assumption)
  apply (auto simp add:wpsimps)[1]
  apply (rule_tac voters=mp and x="Address x" in obtain_voters, assumption)
  apply (auto simp add:wpsimps)[1]
  apply (vcg | solve<auto simp add:wpsimps>)+
  apply (auto simp add:wpsimps intro!: wpthrow)
  apply (rule_tac voters=mp and x="Address x" in obtain_voters, assumption)
  apply (auto simp add:wpsimps)[1]
  apply (rule_tac mp="mp
      (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 x := storage_data.Array [storage_data.Value (Uint (wa + w)), storage_data.Value
(Bool False), storage_data.Value (Address x), storage_data.Value (Uint va)])" in sum_votesI)
  apply (auto simp add:wpsimps intro!: wpthrow)
  apply (rule inv_1;assumption)
  apply (rule_tac voters=mp and x="Address x" in obtain_voters, assumption)
  apply (auto simp add:wpsimps)[1]
  apply (rule_tac voters=mp and x="Address x" in obtain_voters, assumption)
  apply (auto simp add:wpsimps)[1]
  apply (rule_tac voters=mp and x="Address x" in obtain_voters, assumption)
  apply (auto simp add:wpsimps)[1]
  apply (rule_tac voters=mp and x="Address x" in obtain_voters, assumption)
  apply (auto simp add:wpsimps)[1]
  apply (rule_tac voters=mp and x="Address x" in obtain_voters, assumption)
  apply (auto simp add:wpsimps)[1]
done
next
show "∧call proposal. (∧x h r. effect (call x) h r ⇒ vcond x h r) ⇒
  effect (vote call proposal) s r ⇒ inv_state sum_votes s ⇒ post s r sum_votes (K True) (K (K
(K (K True))) proposal)"
  unfolding vote_def
  apply (erule post_exc_true, erule_tac post_wp)
  unfolding inv_state_def
  apply (wp)
  apply (wp)
  apply (wp)
  apply (rule_tac voters=mp and x="Address msg_sender" in obtain_voters, blast)
by (auto simp add:wpsimps intro!: wprules)+
next
show "∧call winningProposalu. (∧x h r. effect (call x) h r ⇒ vcond x h r) ⇒
  effect (winningProposal call winningProposalu) s r ⇒ inv_state sum_votes s ⇒ post s r
sum_votes (K True) (K (K (K (K True))) winningProposalu)"

```

```

unfolding winningProposal_def
apply (erule post_exc_true, erule_tac post_wp)
unfolding inv_state_def
apply (wp)
apply (wp)
apply (wp)
apply (rule_tac voters=mp and x="Address msg_sender" in obtain_voters, blast)
apply (auto simp add:wpsimps intro!:wprules dest!:wpdrules elim!:wperules)
apply (rule_tac iv ="\s'. state.Storage s' = state.Storage s
      ^ (∃x. (Stack s' $$ STR ''p'' = Some (kdata.Value (Uint x))))
      ^ (∃x. (Stack s' $$ STR ''winningVoteCount'' = Some (kdata.Value (Uint x))))
      ^ (∃x. (Stack s' $$ STR ''winningProposalu'' = Some (kdata.Value (Uint x))))" in
wpphile)
  apply (vcg | solve<auto simp add:wpsimps>)+
  apply (auto simp add:wpsimps intro!: wpthrow)
  apply vcg
  apply (rule_tac ya=x in obtain_props, assumption)
  apply (auto simp add:wpsimps intro!: wpthrow)
  apply (vcg | solve<auto simp add:wpsimps>)+
  apply (rule_tac mp=mp and da=a in sum_votesI)
  apply (auto simp add:wpsimps intro!: wpthrow)
done
qed

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 $\equiv \sum ad \in UNIV. unat (valtype.uint (storage_data.vt (x (Address ad))))$ "

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 "( $\forall x y e.$ 
  (fst s) (STR ''ended'') = storage_data.Value (Bool e)  $\wedge$ 
  (fst s) (STR ''pendingReturns'') = storage_data.Map x  $\wedge$ 
  (fst s) (STR ''highestBid'') = storage_data.Value (Uint y)
 $\rightarrow$  (if e = False then ((snd s)  $\geq$  SUMM x + unat y) else (snd s)  $\geq$  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"

```

```

using assms unfolding kdequals_def by simp

lemma kdequals_false:
  assumes "kdequals (rvalue.Value (Address x)) (rvalue.Value (Address y)) = Some (rvalue.Value (Bool
False))"
  shows "x ≠ y"
  using assms unfolding kdequals_def by simp

lemma kdnnot[wpdrules]:
  assumes "kdnnot (rvalue.Value (Bool t)) = Some (rvalue.Value (Bool True))"
  shows "¬ t"
  using assms unfolding kdnnot_def vtnot_def by simp

lemma kdnnot2[wpdrules]:
  assumes "kdnnot x = Some (rvalue.Value (Bool t))"
  shows "x = (rvalue.Value (Bool (¬ t)))"
  using assms unfolding kdnnot_def vtnot_def
  apply (case_tac x)
  apply (auto)
  apply (case_tac x4)
  by auto

lemma kdequals_true_arg_rvalue_true:
  assumes "kdequals yb (rvalue.Value (Bool True)) = Some (rvalue.Value (Bool True))"
  shows "yb = rvalue.Value (Bool True)"
  using assms
  apply (cases yb; simp add:kdequals_def)
  by (case_tac x4; simp add: kdequals_def)

lemma rvalue_equal_kdequals_true:
  assumes "kdequals y x = Some (rvalue.Value (Bool True))"
  shows "y = x"
  using assms unfolding kdequals_def apply (cases y; cases x) apply (auto)
  apply (case_tac x4; case_tac x4a) apply (auto split: if_split if_split_asm)
  done

lemma kdequals_true_arg_rvalue_false:
  assumes "kdequals (rvalue.Value (Bool xe)) (rvalue.Value (Bool False)) = Some (rvalue.Value (Bool
True))"
  shows "xe = False" using assms unfolding kdequals_def by simp

lemma kdplus_safe_storage_none[wpsimps]:
  shows "kdplus_safe x (rvalue.Storage p) = None"
  unfolding kdplus_safe_def by simp

lemma kdplus_safe_storage_some_false[wpdrules]:
  assumes "kdplus_safe x (rvalue.Storage p) = Some a"
  shows "False"
  using assms by (simp add:wpsimps)

lemma storage_data_value_is_value [wpsimps]:
  assumes "storage_data.Value (Uint xc) = y"
  shows "storage_data.is_Value y"
  using assms
  by (auto)

lemma kdplussafe_sint [wpsimps]:
  assumes "∀a. ∃y. x a = storage_data.Value (Uint y)"

```

obtains y where $"x \ a = \text{storage_data.Value (Uint } y)"$ using `assms` by `auto`

8.8 Verifying an Invariant (SimpleAuction)

`lemma notwp[wperules]: "¬ wp m P E s ⇒ wp m P E s ⇒ R" using notE by simp`

`lemma kdequals_rvalue_false_larg_inf: assumes "kdequals yi (rvalue.Value (Bool i)) = Some (rvalue.Value (Bool False))"`
`shows "yi = (rvalue.Value (Bool (¬ i)))"`
`using assms unfolding kdequals_def apply (cases yi) apply (auto) apply (case_tac x4) apply (auto)`
`done`

`lemma add_right_lesseq:`
`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"`
`using assms sum_addr[of _ this] by simp`

`lemma sub_right_lesseq:`
`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"`
`using assms sum_addr[of _ msg_sender] by simp`

`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"`
`using assms sum_addr[of _ msg_sender] by simp`

`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 -`
`from sum_addr[of _ msg_sender] have " (∑ ad|ad ∈ UNIV ∧ ad ≠ msg_sender. unat (valtype.uint (storage_data.vt (xb (Address ad))))) +`
`(unat (valtype.uint (storage_data.vt (xb (Address msg_sender)))) - unat ya ≤ Balances s this +`
`unat msg_value - unat ya"`
`using assms(1,2, 3) by auto`
`moreover have "unat (valtype.uint (storage_data.vt (storage_data.Value (Uint 0)))) ≤ unat (valtype.uint (storage_data.vt (xb (Address msg_sender)))) + unat msg_value - unat ya"`
`using assms(1, 2) unat_add_lem[where ?'a =256] by simp`
`ultimately show ?thesis using sum_addr[of _ msg_sender] by simp`
`qed`

`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 < 2256"`

```

  shows " ( $\sum ad \in UNIV. \text{unat} (\text{valtype.unt} (\text{storage\_data.vt} (\text{if } ad = xc \text{ then } \text{storage\_data.Value} (\text{Unt} (yf + xd)) \text{ else } \text{xb} (\text{Address } ad))))))$ 
     $\leq$  Balances s this"

```

proof-

```

  from sum_addr[of _ xc] have " $(\sum ad / ad \in UNIV \wedge ad \neq xc. \text{unat} (\text{valtype.unt} (\text{storage\_data.vt} (\text{xb} (\text{Address } ad)))))) +$ 
    ( $\text{unat} (\text{valtype.unt} (\text{storage\_data.vt} (\text{xb} (\text{Address } xc)))))) + \text{unat } xd \leq$  Balances s this" using assms
(1) by simp
  moreover have " $\text{unat} (\text{valtype.unt} (\text{storage\_data.vt} (\text{storage\_data.Value} (\text{Unt} (yf + xd)))))) \leq$   $\text{unat} (\text{valtype.unt} (\text{storage\_data.vt} (\text{xb} (\text{Address } xc)))) + \text{unat } xd$ "
  using assms (2-3) unat_add_lem[where ?'a =256] by auto
  ultimately show ?thesis using sum_addr[of _ xc] by simp
qed

```

lemma highestbid_update:

```

  assumes "SUMM xb  $\leq$  Balances s this"
  and "xb (Address xc) = storage_data.Value (Unt yf)"
  and " $\text{unat } yf + \text{unat } xd < 2^{256}$ "
  and " $xd < \text{msg\_value}$ "
  shows " ( $\sum ad \in UNIV. \text{unat} (\text{valtype.unt} (\text{storage\_data.vt} (\text{if } ad = xc \text{ then } \text{storage\_data.Value} (\text{Unt} (yf + xd)) \text{ else } \text{xb} (\text{Address } ad))))))$ 
     $\leq$  Balances s this +  $\text{unat } \text{msg\_value}$ "

```

proof-

```

  from sum_addr[of _ xc] have " $(\sum ad / ad \in UNIV \wedge ad \neq xc. \text{unat} (\text{valtype.unt} (\text{storage\_data.vt} (\text{xb} (\text{Address } ad)))))) +$ 
    ( $\text{unat} (\text{valtype.unt} (\text{storage\_data.vt} (\text{xb} (\text{Address } xc)))))) + \text{unat } \text{msg\_value} \leq$  Balances s this +  $\text{unat } \text{msg\_value}$ " using assms (1) by simp
  moreover have " $\text{unat } xd < \text{unat } \text{msg\_value}$ " using assms(4) using Word.unat_mono[of xd] by simp
  ultimately have " $(\sum ad / ad \in UNIV \wedge ad \neq xc. \text{unat} (\text{valtype.unt} (\text{storage\_data.vt} (\text{xb} (\text{Address } ad)))))) +$ 
     $\text{unat} (yf + xd) \leq$  Balances s this +  $\text{unat } \text{msg\_value}$ " using assms(2,3) unat_add_lem[where ?'a =256]
  by simp
  then show ?thesis using sum_addr[of _ xc] by simp
qed

```

abbreviation (in Solidity) post_const where

```

"post_const (bidding_time::256 word) beneficiary start_state return_value end_state  $\equiv$ 
  state.Storage end_state this STR ''Beneficiary'' = storage_data.Value (Address beneficiary)  $\wedge$ 
  state.Storage end_state this STR ''auctionEndTime'' = storage_data.Value (Unt (timestamp +
  bidding_time))"

```

abbreviation (in Solidity) post_bid where

```

"post_bid start_state return_value end_state  $\equiv$ 
  state.Storage end_state this STR ''highestBidder'' = storage_data.Value (Address msg_sender)  $\wedge$ 
  state.Storage end_state this STR ''highestBid'' = storage_data.Value (Unt msg_value)  $\wedge$ 
   $\neg$  (state.Storage start_state this STR ''highestBid'' = storage_data.Value (Unt 0))
 $\longrightarrow$  ( $\forall mo mn b h prn \text{ pro.}$ 
  state.Storage start_state this STR ''pendingReturns'' = storage_data.Map mo  $\wedge$ 
  state.Storage end_state this STR ''pendingReturns'' = storage_data.Map mn  $\wedge$ 
  state.Storage start_state this STR ''highestBidder'' = storage_data.Value (Address b)  $\wedge$ 
  state.Storage start_state this STR ''highestBid'' = storage_data.Value (Unt h)  $\wedge$ 
  mn (Address b) = storage_data.Value (Unt prn)  $\wedge$ 
  mo (Address b) = storage_data.Value (Unt pro)
 $\longrightarrow$  prn = pro + h)"

```

abbreviation (in Contract) post_bid2 where

```

"post_bid2 start_state return_value end_state  $\equiv$  True"

```

lemma kdless_true_arg_true: assumes "kdless (rvalue.Value (Unt xd)) (rvalue.Value (Unt msg_value)) = Some (rvalue.Value (Bool True))"

```

  shows " $xd < \text{msg\_value}$ "
  using assms unfolding kdless_def vtless_def by simp

```

```

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 -
  show " $\wedge$ call. effect (constructor call si ad) s r  $\implies$ 
    ( $\wedge$ x h r. effect (call x) h r  $\implies$  vcond x h r)  $\implies$  post s r pr_less_Balance (K True) (post_const
  si ad)"
    unfolding constructor_def
    apply (erule post_exc_true, erule_tac post_wp)
    unfolding inv_state_def
  apply (wp wprules: pr_less_BalanceI | auto simp add:wpsimps dest:sym)+
  done

  show "  $\wedge$ call.
    effect (bid call) s r  $\implies$ 
    ( $\wedge$ x h r. effect (call x) h r  $\implies$  vcond x h r)  $\implies$ 
    inv_state pr_less_Balance s  $\implies$ 
    post s r pr_less_Balance (K True) post_bid"
    unfolding bid_def
    apply (erule post_exc_true, erule_tac post_wp)
    unfolding inv_state_def
  apply (vcg wprules: pr_less_BalanceI/auto simp add:wpsimps)+
  apply (rule_tac[!] x="mp" and a="Address ada" in kdplussafe_sint, assumption)
  apply (vcg wprules: pr_less_BalanceI/auto simp add: wpsimps sum_def_spec highestbid_update dest:sym
  dest!: rvalue_equal_kdequals_true kdnot2 kdequals_rvalue_false_larg_inf kdequals_true_arg_rvalue_true
  kdless_true_arg_true)+
  done

  show " $\wedge$ call. effect (withdraw call) s r  $\implies$  ( $\wedge$ x h r. effect (call x) h r  $\implies$  vcond x h r)  $\implies$ 
  inv_state pr_less_Balance s  $\implies$  post s r pr_less_Balance (K True) (K (K (K True)))"
    unfolding withdraw_def
    apply (erule post_exc_true, erule_tac post_wp)
    unfolding inv_state_def
  apply (wp | auto | simp add:wpsimps)+
  apply (vcg | auto simp add:wpsimps)+
  apply (frule_tac x = "Address msg_sender" in spec)
  apply (erule_tac exE)
  apply (wp | auto simp add:wpsimps)+
  apply (case_tac "msg_sender = this")
  apply (auto simp add:wpsimps add_right_lessineq sub_right_lessineq)
  apply (rule_tac pr_less_BalanceI)
  apply (auto simp add:wpsimps add_right_lessineq selfcall_balance_ineq)
  apply (rule_tac pr_less_BalanceI)
  apply (auto simp add:wpsimps sub_right_lessineq notselfcall_balance_ineq)
  apply vcg
  apply (rule_tac pr_less_BalanceI)
  apply (auto simp add:wpsimps)
  apply vcg
  apply (rule_tac pr_less_BalanceI)
  apply (auto simp add:wpsimps)
  apply vcg
  done

  show "  $\wedge$ call.
    effect (auctionEnded call) s r  $\implies$  ( $\wedge$ x h r. effect (call x) h r  $\implies$  vcond x h r)  $\implies$  inv_state
  pr_less_Balance s  $\implies$  post s r pr_less_Balance (K True) (K (K (K True)))"
    unfolding auctionEnded_def

```

```
apply (erule post_exc_true, erule_tac post_wp)
unfolding inv_state_def
apply (vcg)
apply (rule_tac pr_less_BalanceI)
apply (auto simp add:wpsimps split:if_split_asm dest:sym)
apply (case_tac "ad=this")
apply (auto simp add:wpsimps dest: kdequals_true_arg_rvalue_true kdequals_true_arg_rvalue_false)
apply(vcg)
apply (rule_tac pr_less_BalanceI)
apply (auto simp add:wpsimps split:if_split_asm dest:sym)
apply(vcg)
done
qed
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.