Type Safety for Solidity

Billy Thornton Diego Marmsoler

November 16, 2026
Department of Computer Science, University of Exeter, Exeter, UK
{bt319, d.marmsoler}@exeter.ac.uk

Abstract

This development studies type consistency and its preservation for the Isabelle/HOL semantics of Solidity. It defines consistency properties for values, memory, storage, environments, and states, and establishes supporting results for initialisation and copying. Within the stated locale assumptions, the statement preservation theorem shows that successful execution from a type-consistent, fully initialised environment satisfies the state invariant. Concrete examples illustrate the assumptions and the use of the theorem. The underlying Solidity semantics is imported as a separate session.
Keywords: Type safety, Smart contracts, Solidity, Program verification, Isabelle/Solidity

Contents

1 Introduction **3**

2 Iteration lemmas (TypeSafe_Support) **5**

3 Locations and sublocations (TypeSafe_Hashing_Subs) **5**

4 Type consistency for value types (TypeSafe_Base_Types) **9**

5 Type consistency for memory (TypeSafe_Memory) **10**

6 Type consistency for storage (TypeSafe_Storage) **20**

7 Type consistency for environments and states (TypeSafe_Def) **27**

8 Storage-to-storage copies (TypeSafe_Copies) **33**

9 Memory initialisation (TypeSafe_Memory_Init) **35**

10 Compatibility of storage and memory types (TypeSafe_Mem_Sto_Comp) **38**

11 Copies between memory and storage (TypeSafe_Memory_Copies) **38**

12 Environment initialisation and declarations (TypeSafe_Environment) **49**

13 Declarations (TypeSafe_Decl) **51**

14 Expression preservation (TypeSafe_Expressions) **56**

15 Supporting lemmas for statement preservation (TypeSafe_Statement_Support) **59**

16 Preservation for assignments (TypeSafe_Assign) **64**

17 Statement preservation (TypeSafe) **68**

18 Examples (TypeSafe_Examples) **68**

1 Introduction

This development builds on the Isabelle/HOL formalisation of Solidity by Marmsoler and Brucker [1]. The formalization is described in two accompanying papers [2, 3]. The rest of this document is automatically generated from the formalization in Isabelle/HOL, i.e., all content is checked by Isabelle. The structure follows the theory dependencies (see Figure 1).

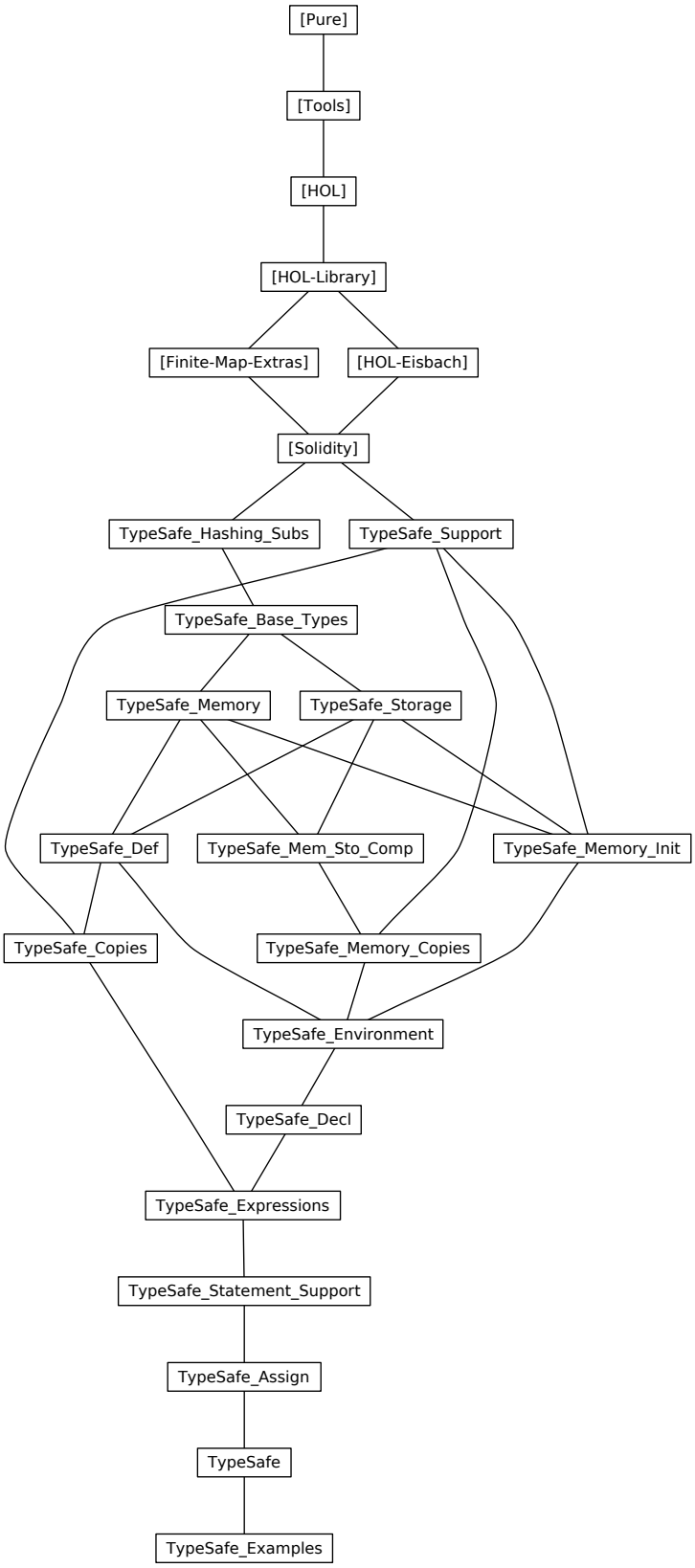


Figure 1: The Dependency Graph of the Isabelle Theories.

2 Iteration lemmas (TypeSafe_Support)

theory *TypeSafe_Support*
  imports *"Solidity.Solidity_Main"*
begin

Induction and invariant principles for *iter'* and *iter*, which drive the array-copying and initialisation functions of the semantics.

lemma *iter'_induct:*
  assumes *"⋀v'. iter' f v 0 = Some v' ⟹ P 0 v'"*
    "⋀x v''. (∃v'. iter' f v x = Some v' ∧ P x v' ∧ f x v' = Some v'') ⟹ iter' f v (Suc x) = Some v'' ⟹ P (Suc x) v''"
  shows *"iter' f v x = Some v' ⟹ P x v'"*
⟨*proof*⟩

lemma *iter'_invariant:*
  assumes *run: "iter' f v x = Some v'"*
    and *init: "I v"*
    and *step: "⋀i v1 v2. I v1 ⟹ f i v1 = Some v2 ⟹ I v2"*
  shows *"I v'"*
⟨*proof*⟩

lemma *iter'_indexed_invariant:*
  assumes *run: "iter' f v x = Some v'"*
    and *init: "P 0 v"*
    and *step: "⋀i v1 v2. P i v1 ⟹ f i v1 = Some v2 ⟹ P (Suc i) v2"*
  shows *"P x v'"*
⟨*proof*⟩

lemma *iter'_indexed_type_writes:*
  fixes *loc :: "nat ⇒ String.literal"*
  assumes *run: "iter' f v x = Some v'"*
    and *old: "⋀i j v1 v2. j < i ⟹ f i v1 = Some v2*
      ⟹ accessTypeStore (loc j) v2 = accessTypeStore (loc j) v1"
    and *new: "⋀i v1 v2. f i v1 = Some v2 ⟹ accessTypeStore (loc i) v2 = Some tp"*
  shows *"∀j<x. accessTypeStore (loc j) v' = Some tp"*
⟨*proof*⟩

lemma *iter_ind:*
  assumes *"iter f v 0 = v ⟹ P 0 v"*
    "⋀x v''. (∃v'. iter f v x = v' ∧ P x v' ∧ f x v' = v'') ⟹ iter f v (Suc x) = v'' ⟹ P (Suc x) v''"
  shows *"iter f v x = v' ⟹ P x v'"*
⟨*proof*⟩

lemma *iter_indexed_type_writes:*
  fixes *loc :: "nat ⇒ String.literal"*
  assumes *run: "iter f v x = v'"*
    and *old: "⋀i j v1 v2. j < i ⟹ f i v1 = v2*
      ⟹ accessTypeStore (loc j) v2 = accessTypeStore (loc j) v1"
    and *new: "⋀i v1 v2. f i v1 = v2 ⟹ accessTypeStore (loc i) v2 = Some tp"*
  shows *"∀j<x. accessTypeStore (loc j) v' = Some tp"*
⟨*proof*⟩

lemma *updateTypedStore_root_type[simp]:*
  "accessTypeStore loc (updateTypedStore loc v tp m) = Some tp"
  ⟨*proof*⟩

end

3 Locations and sublocations (TypeSafe_Hashing_Subs)

theory *TypeSafe_Hashing_Subs*

imports *"Solidity.Solidity_Main" "HOL-Library.Sublist"*
begin

3.1 Hashing lemmas

fun *location_depth :: "location ⇒ nat"* **where**
"location_depth s = length (filter (λx. x = CHR ''.'') (String.explode s))"

lemma *explode_STR_dot: "String.explode (STR ''.'') = [CHR ''.'']"*
⟨*proof*⟩

lemma *location_depth_hash_property:*
"location_depth (hash ls suf) = location_depth ls + location_depth suf + 1"
⟨*proof*⟩

lemma *hash_adds_depth:*
assumes *"CHR ''.'' ∉ set (String.explode suf) "*
shows *"location_depth ((hash ls suf)) = location_depth (ls) + 1"*
⟨*proof*⟩

Hashing a location with any suffix yields a different location.

lemma *hash_inequality: "∀ suffix. (hash ls suffix) ≠ ls "*
⟨*proof*⟩

lemma *hash_version_inequality: "∀ suffix. (hash_version ls suffix) ≠ ls "*
⟨*proof*⟩

lemma *hash_suffixes_associative: "(hash (hash ls suffix1) suffix2) = (hash ls (hash suffix1 suffix2))"*
⟨*proof*⟩

lemma *not_equal_string:*
assumes *"s1≠s2"*
shows *"String.explode s1 ≠ String.explode s2"*
⟨*proof*⟩

Distinct prefixes hash to distinct locations.

lemma *hash_never_equal_prefix:*
assumes *"s1 ≠ s2"*
shows *"∀ iter1. (hash s1 iter1) ≠ hash s2 iter1"*
⟨*proof*⟩

Distinct base locations hash to distinct locations.

lemma *hash_never_equal_suffix:*
assumes *"s1 ≠ s2"*
shows *"∀ iter1. (hash iter1 s1) ≠ hash iter1 s2"*
⟨*proof*⟩

definition *LSubPrefL2::"location ⇒ location ⇒ bool"*
where *"LSubPrefL2 moreSpecificThanParent parent =*
((∃ i. moreSpecificThanParent = (hash parent i)) ∨ moreSpecificThanParent = parent)"

lemma *subPrefCannotBeInt:*
assumes *"CHR ''.'' ∉ set(String.explode l)"*
shows *"¬(∃ i. l = i + (STR ''.'' + stl2))"*
⟨*proof*⟩

lemma *hash_int_prefix:*
assumes *"hash destl i ≠ hash destl x"*
and *"CHR ''.'' ∉ set(String.explode i)"*
and *"CHR ''.'' ∉ set(String.explode x)"*
shows *"¬ LSubPrefL2 (hash destl i) (hash destl x)"*
⟨*proof*⟩

lemma *Not_Sub_More_Specific:*
assumes *"¬ LSubPrefL2 destl' destl"*
shows *"¬ LSubPrefL2 destl' (hash destl x)"*
⟨*proof*⟩

lemma *suffix_eq_harder:*
assumes *"b @ s @ a = d @ s @ c"*
and *"b ≠ d"*
shows *"strict_prefix b d ∨ strict_prefix d b"*
⟨*proof*⟩

lemma *Mutual_NonSub_SpecificNonSub:*
assumes *"¬LSubPrefL2 destl' destl ∧ ¬LSubPrefL2 destl destl'"*
shows *"¬LSubPrefL2 (hash destl' x) destl"*
⟨*proof*⟩

lemma *hashesIntSame:*
assumes *"hash destl (ShowL$_{nat}$ i) = y"*
and *"hash destl (ShowL$_{nat}$ i') = y"*
shows *"i = i'"* ⟨*proof*⟩

lemma *hashesInts:*
assumes *"hash (hash destl (ShowL$_{nat}$ i)) z = y"*
and *"hash (hash destl (ShowL$_{nat}$ i')) z' = y"*
shows *"i = i'"* ⟨*proof*⟩

lemma *hashesAssociative:*
assumes *"hash (hash destl i) z = y"*
and *"hash (hash destl i') z' = y"*
and *"CHR ''.'' ∉ set(String.explode i)"*
and *"CHR ''.'' ∉ set(String.explode i')"*

shows *"i = i'"* ⟨*proof*⟩

lemma *LSubPrefL2_refl: "LSubPrefL2 l l"*
⟨*proof*⟩

lemma *LSubPrefL2_hash: "LSubPrefL2 (hash l k) l"*
⟨*proof*⟩

lemma *LSubPrefL2_hash_mono:*
assumes *"LSubPrefL2 a b"*
shows *"LSubPrefL2 (hash a k) b"*
⟨*proof*⟩

lemma *LSubPrefL2_trans:*
assumes *"LSubPrefL2 a b"* **and** *"LSubPrefL2 b c"*
shows *"LSubPrefL2 a c"*
⟨*proof*⟩

A location lies below at most one dot-free index of a parent.

lemma *LSubPrefL2_hash_unique:*
assumes *"LSubPrefL2 l (hash p i)"* **and** *"LSubPrefL2 l (hash p j)"*
and *"CHR ''.'' ∉ set (String.explode i)"* **and** *"CHR ''.'' ∉ set (String.explode j)"*
shows *"i = j"*
⟨*proof*⟩

A location lies below at most one index of an array.

lemma *LSubPrefL2_index_unique:*
assumes *"LSubPrefL2 l (hash p (ShowL$_{nat}$ i))"*
and *"LSubPrefL2 l (hash p (ShowL$_{nat}$ j))"*
shows *"i = j"*

⟨*proof*⟩

lemma *readLintNotEqual:*
assumes "(ReadL$_{nat}$ suffix')<(ReadL$_{nat}$ suffixa)"
shows "suffix' ≠ suffixa" ⟨*proof*⟩

lemma *checkUIntIncreaseB:*
assumes "(ReadL$_{int}$ v ≥ 0 ∧ ReadL$_{int}$ v < 2^b)"
and "b≤b2"
shows "(ReadL$_{int}$ v ≥ 0 ∧ ReadL$_{int}$ v < 2^b2)"
⟨*proof*⟩

lemma *ShowLnat_ReadLint_inverse:*
assumes "0 ≤ ReadL$_{int}$(kv)"
and "ShowL$_{int}$ (ReadL$_{int}$ kv) = kv"
shows "(ShowL$_{nat}$ (nat (ReadL$_{int}$ kv))) = kv"
⟨*proof*⟩

lemma *notNoneUpdate:*
assumes "st' = st⦇Stack := updateStore l v (stack st), Gas:=g⦈"
shows "accessStore l (Stack st') = Some v" ⟨*proof*⟩

lemma *MemLSubPrefL2_specific_imps_general:*
assumes "LSubPrefL2 (hash p y) (ShowL$_{nat}$ tloc)"
shows "LSubPrefL2 p (ShowL$_{nat}$ tloc)"
⟨*proof*⟩

3.2 Miscellaneous lemmas

lemma *checkSIntIncreaseB:*
assumes "(ReadL$_{int}$ v ≥ -(2^(b-1)) ∧ ReadL$_{int}$ v < 2^(b-1))"
and "b≤b2"
shows "(ReadL$_{int}$ v ≥ -(2^(b2-1)) ∧ ReadL$_{int}$ v < 2^(b2-1))"
⟨*proof*⟩

lemma *checkUIntToSInt:*
assumes "(ReadL$_{int}$ v ≥ 0 ∧ ReadL$_{int}$ v < 2^b)"
and "b<b2"
shows "(ReadL$_{int}$ v ≥ -(2^(b2-1)) ∧ ReadL$_{int}$ v < 2^(b2-1))"
⟨*proof*⟩

3.3 Additional Hash Helper Lemmas

Hash length and structure properties.

lemma *hash_contains_dot:*
"CHR ''.'' ∈ set (String.explode (hash a b))"
⟨*proof*⟩

lemma *hash_flatten_right:*
"hash a (hash b c) = hash (hash a b) c"
⟨*proof*⟩

lemma *MemLSubPrefTransitive:*
assumes "LSubPrefL2 loc (ShowL$_{nat}$ tloc)"
and "LSubPrefL2 loc p'"
shows "LSubPrefL2 p' (ShowL$_{nat}$ tloc)"
⟨*proof*⟩

3.4 Stack, calldata, and memory locations are below the top location

definition *lessThanTopLocs :: "('a, 'v) store_scheme ⇒ bool"* **where**
"lessThanTopLocs st = ((∀ tloc loc. tloc ≥ (Toploc st) ∧ LSubPrefL2 loc (ShowL$_{nat}$ tloc) ⟶ accessStore loc st = None)

∧ (∀ loc y. accessStore loc st = Some y ⟶ (∃ tloc. tloc<(Toploc st) ∧ LSubPrefL2 loc (ShowL$_{nat}$ tloc))))"

end

4 Type consistency for value types (TypeSafe_Base_Types)

theory *TypeSafe_Base_Types*
  imports *TypeSafe_Hashing_Subs*
begin

4.1 Ensuring that String *valuetype* conform to their associated datatype *Valuetypes.types*

primrec *typeCon :: "types ⇒ valuetype ⇒ bool"*
  where *"typeCon (TSInt x) v = checkSInt x v"*
  | *"typeCon (TUInt x) v = checkUInt x v"*
  | *"typeCon (TBool) v = checkBool v"*
  | *"typeCon (TAddr) v = checkAddress v"*

lemma *typeConNoDots:*
  assumes *"typeCon t v"*
  shows *"CHR ''.'' ∉ set(String.explode v)"*
⟨*proof*⟩

lemma *typeCon_no_sublocation_prefix:*
  assumes *"typeCon x1 i"*
  assumes *"hash destl i ≠ hash destl x"*
  shows *"¬ LSubPrefL2 (hash destl i) (hash destl x)"*
  ⟨*proof*⟩

lemma *addBalance_Bal:*
  assumes *"addBalance ad val acc = Some acc'"*
  shows *"Bal (acc' ad) = ShowL$_{int}$ (ReadL$_{int}$ (Bal (acc ad)) + ReadL$_{int}$ val)"*
  ⟨*proof*⟩

lemma *subBalance_Bal:*
  assumes *"subBalance ad val acc = Some acc'"*
  shows *"Bal (acc' ad) = ShowL$_{int}$ (ReadL$_{int}$ (Bal (acc ad)) - ReadL$_{int}$ val)"*
  ⟨*proof*⟩

A successful transfer keeps every balance a valid *uint256*.

lemma *transfer_Bal_typeCon:*
  assumes *"transfer ads addr val acc = Some acc'"*
    and *"∀ a. typeCon (TUInt b256) (Bal (acc a))"*
  shows *"typeCon (TUInt b256) (Bal (acc' a))"*
⟨*proof*⟩

lemma *transfer_subRead:*
  assumes *"transfer ads addr val acc = Some acc'"*
    and *"addr ≠ ads"*
    and *"typeCon (TUInt b256) val"*
    and *"typeCon (TUInt b256) (Bal (acc ads))"*
  shows *"(Bal (acc' ads)) = ShowL$_{int}$(ReadL$_{int}$ (Bal (acc ads)) - ReadL$_{int}$ val)"*
⟨*proof*⟩

lemma *transfer_addRead:*
  assumes *"transfer ads addr val acc = Some acc'"*
    and *"addr ≠ ads"*
    and *"typeCon (TUInt b256) val"*
    and *"typeCon (TUInt b256) (Bal (acc addr))"*
  shows *"(Bal (acc' addr)) = ShowL$_{int}$(ReadL$_{int}$ (Bal (acc addr)) + ReadL$_{int}$ val)"*

⟨*proof*⟩

lemma *transfer_sameRead:*
  assumes *"transfer ads addr val acc = Some acc'"*
    and *"addr = ads"*
    and *"typeCon (TUInt b256) val"*
    and *"typeCon (TUInt b256) (Bal (acc ads))"*
  shows *"(Bal (acc' ads)) = (Bal (acc ads))"*
⟨*proof*⟩

4.2 Creating Signed and Unsigned integers

lemma *checkSIntCreate:*
  shows *"checkSInt (b) (createSInt b v)"* ⟨*proof*⟩

lemma *checkUIntCreate:*
  shows *"checkUInt (b) (createUInt b v)"* ⟨*proof*⟩

4.3 Compatible types will both satisfy the consistency of a value

If two types are compatible and a value is consistent with the first, it is consistent with the second.

lemma *SameCompTypeCon:*
  assumes *"comp t t'"*
    and *"typeCon t v"*
  shows *"typeCon t' v"*
⟨*proof*⟩

4.4 Initial values

When a new location is created in Memory, Storage or calldata, or a new variable is created then a default initial value is used. These values must satisfy *typeCon*, which is what the following lemma demonstrates.

lemma *ivalTypeCon:*
  assumes *"ival t' = v'"*
  shows *"typeCon t' v'"*
⟨*proof*⟩

4.5 Convert is typesafe

Conversion between types preserves type consistency. If the string value was consistent with the initial type and a successful conversion is performed then the result must be consistent with the new type.

lemma *typeSafeConvert:*
  assumes *"typeCon t2 v"*
    and *"convert t2 t' v = Some(v)"*
  shows *"typeCon t' v"*
⟨*proof*⟩

end

5 Type consistency for memory (TypeSafe_Memory)

theory *TypeSafe_Memory*
  imports *TypeSafe_Base_Types*
begin

fun *extractType :: "mtypes ⇒ mtypes option"* **where**
  "extractType (MTArray len arr) = Some arr"
| "extractType (MTValue v) = None"

definition *AllocatedMem_between :: "memoryT ⇒ memoryT ⇒ location set"*
  where

```
    "AllocatedMem_between st st' = ({ShowL_nat l | l. Toploc st ≤ l ∧ l < Toploc st'})
                                  ∪ fset(fmdom (Mapping (st')) - fmdom (Mapping (st)))"

primrec CompMemJustType::"mtypes ⇒ mtypes ⇒ bool"
  where
    "CompMemJustType (MTValue typ) mem = (mem = MTValue typ)"
  | "CompMemJustType (MTArray len arr) mem = (mem = (MTArray len arr) ∨ CompMemJustType arr mem)"

primrec CompMemType :: "memoryT ⇒ nat ⇒ mtypes ⇒ mtypes ⇒ location ⇒ location ⇒ bool"
  where
    "CompMemType mem len (MTValue typ) childtp parentloc childloc = (childtp = (MTValue typ) ∧ (∃i<len.
(hash parentloc (ShowL_nat i)) = childloc))"
  | "CompMemType mem len (MTArray len' arr) childtp parentloc childloc = (∃i<len. ∃l. accessStore (hash
parentloc (ShowL_nat i)) mem = Some (MPointer l) ∧
                                                                    ((l = childloc ∧ (MTArray
len' arr) = childtp) ∨ CompMemType mem len' arr childtp l childloc)
                                                                  )"
```

All memory locations occupied by a value of the given memory type, starting at a base location and following array pointers in the given memory.

```
fun GetAllMemoryLocations :: "mtypes ⇒ location ⇒ memoryT ⇒ location set"
  where
    "GetAllMemoryLocations (MTValue _) base_loc _ = {base_loc}"
  | "GetAllMemoryLocations (MTArray len struct) base_loc mem =
     (⋃i∈{0..<len}.
       (case accessStore (hash base_loc (ShowL_nat i)) mem of
          Some (MPointer ptr) ⇒ {(hash base_loc (ShowL_nat i))} ∪ GetAllMemoryLocations struct ptr mem
        | Some (MValue _) ⇒ {(hash base_loc (ShowL_nat i))} ∪ GetAllMemoryLocations struct (hash
base_loc (ShowL_nat i)) mem
        | None ⇒ {}))"
```

5.1 Memory and calldata conformity

`MCon` checks that the sub-locations of a memory or calldata structure conform to their types. Starting at a location, it checks all sub-locations. For an array, every element is checked recursively; for a value type, the string stored at the location is checked against its base type.

Locations stored in the `Denvalue` for memory references already denote the target of the pointer, following the convention of `expressions_with_gas.msel`. For example, if a memory array stored at `0.0` has pointer value `7.7`, the environment stores `7.7`. Hence `MCon` does not dereference its location argument.

```
primrec MCon :: "mtypes ⇒ memoryT ⇒ location ⇒ bool"
  where
    "MCon (MTValue typ) msto loc =(case (accessStore loc msto) of
                                  (Some (MValue t)) ⇒ (typeCon typ t)
                                    | (Some(MPointer t)) ⇒ False
                                    | None ⇒ False)"

|"MCon (MTArray len arr) msto loc = (len>0 ∧ (∀i < len.
                                              (case (accessStore (hash loc (ShowL_nat i)) msto) of
                                               (Some (MPointer loc2)) ⇒
                                                      (case arr of (MTArray len' arr') ⇒ (MCon arr
msto (loc2))
                                                                 | MTValue val ⇒ False)
                                                | Some(MValue val) ⇒
                                                  (case arr of (MTValue typ) ⇒ (MCon arr msto (hash
loc (ShowL_nat i)))
                                                     | (MTArray l a) ⇒ False)
                                                | None ⇒ False))
                                  ∧ (∃p. accessStore loc msto = Some (MPointer p) ∨ accessStore loc
msto = None))"

primrec MConAccessSame:: "location ⇒ location ⇒ mtypes ⇒ memoryT ⇒ memoryT ⇒ bool"
  where
```

```
    "MConAccessSame l1 l2 (MTValue t) m1 m2 = (∃ val. accessStore l1 m1 = Some (MValue val) ∧ (∃ val'.
accessStore l2 m2 = Some (MValue val')))"
  | "MConAccessSame l1 l2 (MTArray len t) m1 m2 = (∀ i1<len. ∀ i2<len. (∀ l. accessStore (hash l1
(ShowL_nat i1)) m1 = Some (MPointer l) ⟶
                                                                    (∃ l'. accessStore (hash l2 (ShowL_nat
i2)) m2 = Some (MPointer l')
                                                                    ∧ MConAccessSame l l' t m1 m2)))"
```

The set of all memory locations reachable from the memory references of an environment and stack.

```
fun ReachableMem :: "environment ⇒ stack ⇒ memoryT ⇒ location set"
  where
    "ReachableMem env st mem = (⋃ (t, l) ∈ fset(fmran (Denvalue env)).
         (case l of
           Stackloc loc ⇒
             (case accessStore loc st of
               Some (KMemptr ptr) ⇒
                 (case t of
                   type.Memory struct ⇒ GetAllMemoryLocations struct ptr mem
                    | _ ⇒ {})
               | _ ⇒ {})
           | Storeloc loc ⇒ {}))"


primrec TypedMemSubPref:: "location ⇒ location ⇒ mtypes ⇒ bool"
  where
    "TypedMemSubPref child parent (MTValue t) = (child = parent)"
  | "TypedMemSubPref child parent (MTArray len t) = (∃ i<len. TypedMemSubPref child (hash parent
(ShowL_nat i)) t
      ∨ child = (hash parent (ShowL_nat i)))"

primrec TypedMemSubPrefPtrs:: "memoryT ⇒ nat ⇒ mtypes⇒ location ⇒ location ⇒ bool"
  where
    "TypedMemSubPrefPtrs mem len (MTValue t) parent child = (∃ i<len. (hash parent (ShowL_nat i)) =
child)"
  | "TypedMemSubPrefPtrs mem len (MTArray len' t) parent child = (∃ i<len. ∃ l. accessStore (hash parent
(ShowL_nat i)) mem = Some (MPointer l)
                                                                    ∧ (l = child ∨ TypedMemSubPrefPtrs
mem len' t l child ))"

definition WrittenMem_between :: "memoryT ⇒ memoryT ⇒ location set"
  where
    "WrittenMem_between st st' = {l. accessStore l st ≠ accessStore l (st')}"

definition compMemPtrs :: "stack ⇒ memoryT ⇒ (String.literal, type × denvalue) fmap ⇒ bool" where
  "compMemPtrs st mem denval = (∀ tp1 tp2 l1 l2 stl1 stl2 dloc1.
          (type.Memory tp1, Stackloc l1) |∈| fmran denval ∧ (type.Memory tp2, Stackloc l2) |∈| fmran
denval
          ∧ accessStore l1 st = Some(KMemptr stl1) ∧ accessStore l2 st = Some(KMemptr stl2)
      ⟶ (if stl1 = stl2 then tp1 = tp2 else
            (case tp1 of
              MTValue val ⇒
                (case tp2 of MTValue val2 ⇒ True
                  | MTArray len2 arr2 ⇒ (if TypedMemSubPrefPtrs mem len2 arr2 stl2 stl1 then
CompMemType mem len2 arr2 (MTValue val) stl2 stl1
                                        else True))
             | MTArray len arr ⇒
                (case tp2 of MTValue val2 ⇒ (if TypedMemSubPrefPtrs mem len arr stl1 stl2 then
CompMemType mem len arr (MTValue val2) stl1 stl2
                                          else True)
                  | MTArray len2 arr2 ⇒ (if TypedMemSubPrefPtrs mem len2 arr2 stl2 stl1 then
CompMemType mem len2 arr2 (MTArray len arr) stl2 stl1
                                        else if TypedMemSubPrefPtrs mem len arr stl1 stl2 then
CompMemType mem len arr (MTArray len2 arr2) stl1 stl2
                                        else if (TypedMemSubPrefPtrs mem len arr stl1 dloc1 ∧
```

```
TypedMemSubPrefPtrs mem len2 arr2 stl2 dloc1)
                                                    then (∃dt. CompMemType mem len2 arr2 dt stl2 dloc1
                                                                                    ∧ CompMemType mem len arr dt
stl1 dloc1
                                                                                    )
                                                else True))

)))"

definition compMemPtrs3 :: "stack ⇒ memoryT ⇒ (String.literal, type × denvalue) fmap ⇒ bool" where
  "compMemPtrs3 st mem denval = (∀tp1 tp2 l1 l2 stl1 stl2 dloc1.
    (type.Memory tp1, Stackloc l1) |∈| fmran denval ∧ (type.Memory tp2, Stackloc l2) |∈| fmran denval
    ∧ accessStore l1 st = Some(KMemptr stl1) ∧ accessStore l2 st = Some(KMemptr stl2)
    ⟶ (if stl1 = stl2 then tp1 = tp2 else
      (case (tp1, tp2) of
          (MTValue val, MTValue val2) ⇒ True
        | (MTValue val, MTArray len2 arr2) ⇒ ¬ (TypedMemSubPrefPtrs mem len2 arr2 stl2 stl1) ∨
CompMemType mem len2 arr2 (MTValue val) stl2 stl1
        | (MTArray len arr, MTValue val2) ⇒ ¬ (TypedMemSubPrefPtrs mem len arr stl1 stl2) ∨
CompMemType mem len arr (MTValue val2) stl1 stl2
        | (MTArray len arr, MTArray len2 arr2) ⇒
            if (TypedMemSubPrefPtrs mem len arr stl1 dloc1 ∧ TypedMemSubPrefPtrs mem len2 arr2 stl2
dloc1)
                then (∃dt. CompMemType mem len2 arr2 dt stl2 dloc1 ∧ CompMemType mem len arr dt stl1
dloc1 )
            else True)
      )
    )"


fun mtype_size :: "mtypes ⇒ nat" where
  "mtype_size (MTValue _) = 1"
| "mtype_size (MTArray _ S) = 1 + mtype_size S"
```

5.2 Type relations between arrays and their elements

```
lemma CompMemJustTypes_trns:
  assumes "CompMemJustType tp1 tp2"
    and "CompMemJustType tp1 tp3"
  shows "CompMemJustType tp2 tp3 ∨ CompMemJustType tp3 tp2" ⟨proof⟩

lemma CompMemType_SameAccessAllocate:
  assumes "∀loc. accessStore loc m = accessStore loc (snd (allocate m'))"
    and "CompMemType m len t1 t2 l1 l2"
  shows "CompMemType m' len t1 t2 l1 l2" ⟨proof⟩

lemma CompMemType_imps_CompMemJustType:
  assumes "CompMemType mem len t1 t2 l1 l2"
  shows "CompMemJustType (MTArray len t1) t2" ⟨proof⟩

lemma CompMemType_imps_TypedMemSubPrefPtrs:
  assumes "CompMemType mem len tp tc lp lc"
  shows "TypedMemSubPrefPtrs mem len tp lp lc" ⟨proof⟩

lemma CompMemTypes_asc:
  assumes "CompMemJustType tp1 tp2"
    and "CompMemJustType tp2 tp3"
  shows "CompMemJustType tp1 tp3" ⟨proof⟩

lemma MConArrayPointers:
  assumes "MCon (MTArray len (MTArray len' t)) v' loc"
    and "len >0"
    and "i < len"
  obtains val where "accessStore (hash loc (ShowL_nat i)) v' = Some (MPointer val)"
```

⟨*proof*⟩

lemma *MCon_array_decrement:*
assumes "MCon (MTArray len t) v' loc"
and "len -1 > 0"
shows "MCon (MTArray (len-1) t) v' loc" ⟨*proof*⟩

lemma *MConTypeSplitingArray:*
assumes "∀(i::nat)<(x1::nat).
(case accessStore (hash stl1 (ShowL$_{nat}$ i)) mem of
Some (MPointer loc2) =>
(case tp1 of MTArray len' arr' => MCon tp1 mem loc2
| MTValue tps => False)
| Some (MValue val) =>
(case tp1 of (MTArray n mt) => False
| MTValue tps => MCon tp1 mem (hash stl1 (ShowL$_{nat}$ i)))
| None => False)"
and "∃a b. tp1 = MTArray a b"
and "(i::nat)<x1"
shows "∃p. accessStore (hash stl1 (ShowL$_{nat}$ i)) mem = Some (MPointer p)"
⟨*proof*⟩

lemma *MConSubTypes:*
assumes "MCon (MTArray len t') mem loc"
shows "∀t. CompMemJustType t' t ⟶ (¬MCon t mem loc)" ⟨*proof*⟩

lemma *MConPtrsMustBeSubLocs:*
assumes "MCon (MTArray len t') mem loc"
shows "∀p i. i<len ∧ accessStore (hash loc (ShowL$_{nat}$ i)) mem = Some (MPointer p) ⟶ p ≠ loc"
⟨*proof*⟩

lemma *MConPtrsMustBeSubLocs_CompMemType:*
assumes "MCon (MTArray len arr) mem loc"
shows "∀p t. MCon t mem p ∧ CompMemType mem len arr t loc p ⟶ p ≠ loc" ⟨*proof*⟩

lemma *BothMConImpsNotCompMemType:*
assumes "MCon (MTArray len arr) mem p''"
and "MCon t' mem p''"
shows "¬CompMemType mem len arr t' p'' p''" ⟨*proof*⟩

lemma *MCon_imps_Some:*
assumes "MCon t' mem p'"
shows "∃x i. accessStore p' mem = Some x ∨ accessStore (hash p' (ShowL$_{nat}$ i)) mem = Some x"
⟨*proof*⟩

lemma *MCon_subTypes_imps_noPrnt:*
assumes "MCon (MTArray x t) mem l"
and "MCon (MTArray x' (MTArray x t)) mem l'"
shows "¬CompMemType mem x t (MTArray x' (MTArray x t)) l l'" ⟨*proof*⟩

abbreviation *example_mem_array_bool::memoryT*
where "example_mem_array_bool ≡
(|Mapping = fmap_of_list
[(STR ''1.1.0'', MValue STR ''False''),
(STR ''0.1.0'', MValue STR ''True''),
(STR ''1.0'', MPointer STR ''1.0''),
(STR ''1.1.0.0'', MValue STR ''False''),
(STR ''0.0.0'', MValue STR ''True''),
(STR ''0.0'', MPointer STR ''0.0'')],
Toploc = 1,
Typed_Mapping = fmap_of_list
[(STR ''1.1.0'', MTValue TBool),

```
       (STR ''0.1.0'', MTValue TBool),
       (STR ''1.0'', MTArray 2 (MTValue TBool)),
       (STR ''1.0.0'', MTValue TBool),
       (STR ''0.0.0'', MTValue TBool),
       (STR ''0.0'', MTArray 2 (MTValue TBool))]⟦"
```

lemma *MCon_sub_MTVal_imps_val:*
assumes *"MCon (MTArray len (MTValue t)) mm loc"*
shows *"∀i<len. ∃val. accessStore (hash loc (ShowL$_{nat}$ i)) mm = Some (MValue val)"*
⟨*proof*⟩

lemma *CompTypeRemainsMCon:*
assumes *"CompMemType mem len arr t2 p' stl1"*
and *"MCon (MTArray len arr) mem p'"*
shows *"MCon t2 mem stl1"* ⟨*proof*⟩

lemma *CompMemJustType_refl: "CompMemJustType t t"*
⟨*proof*⟩

Two sub-types of a common type that are both consistent at the same location are equal.

lemma *MCon_CompMemJustType_unique:*
assumes *"CompMemJustType t t1"* **and** *"CompMemJustType t t2"*
and *"MCon t1 mem l"* **and** *"MCon t2 mem l"*
shows *"t1 = t2"*
⟨*proof*⟩

5.3 Consistency under memory changes

lemma *GetAllMemoryLocations_Arry_Contains_Indexs:*
assumes *"MCon (MTArray len2 arr2) mem stl2"*
shows *"∀i<len2. hash stl2 (ShowL$_{nat}$ i) ∈ GetAllMemoryLocations (MTArray len2 arr2) stl2 mem"* ⟨*proof*⟩

lemma *MCon_imps_sub_Mcon:*
assumes *"MCon (MTArray arLen t) srcMem srcl"*
shows *"∀x::nat<arLen. (∀l. accessStore (hash srcl (ShowL$_{nat}$ x)) srcMem = Some (MPointer l) ⟶ MCon t srcMem l)*
∧ (∀val. accessStore (hash srcl (ShowL$_{nat}$ x)) srcMem = Some (MValue val) ⟶ MCon t srcMem (hash srcl (ShowL$_{nat}$ x)))"
⟨*proof*⟩

lemma *MconSameTypeSameAccessWithTyping:*
assumes *"MCon (MTArray x t) m1 l1"*
and *"MCon (MTArray x t) m2 l2"*
shows *"∀i1<x. ∀i2<x. (∀l. accessStore (hash l1 (ShowL$_{nat}$ i1)) m1 = Some (MPointer l)*
⟶ (∃l'. accessStore (hash l2 (ShowL$_{nat}$ i2)) m2 = Some (MPointer l') ∧ MCon t m1 l
∧ MCon t m2 l'))
∧ (∀val . accessStore (hash l1 (ShowL$_{nat}$ i1)) m1 = Some (MValue val) ⟶
(∃val'. accessStore (hash l2 (ShowL$_{nat}$ i2)) m2 = Some (MValue val')∧ MCon t m1 (hash l1 (ShowL$_{nat}$ i1)) ∧ MCon t m2 (hash l2 (ShowL$_{nat}$ i2))))"
⟨*proof*⟩

lemma *SubPtrs_top:*
assumes *"LSubPrefL2 stl2 (ShowL$_{nat}$ (Toploc mem))"*
and *"∀l l'. LSubPrefL2 l (ShowL$_{nat}$ (Toploc mem)) ∧ accessStore l mem' = Some (MPointer l') ⟶ l' = l"*
and *"TypedMemSubPrefPtrs mem' x11 x12 stl2 dloc1"*
shows *"LSubPrefL2 dloc1 (ShowL$_{nat}$ (Toploc mem))"*
⟨*proof*⟩

lemma *TypedMemSubPrefPtrs_trns:*
assumes *"TypedMemSubPrefPtrs mem x2 t2 l2 l3"*
and *"MCon (MTArray x1 t1) mem l1"*

and *"CompMemType mem x1 t1 (MTArray x2 t2) l1 l2"*
shows *"TypedMemSubPrefPtrs mem x1 t1 l1 l3"* ⟨*proof*⟩

lemma *allocateMCon:*
assumes *"MCon t mm loc"*
and *"snd (allocate mm) = m'"*
shows *"MCon t m' loc"*
⟨*proof*⟩

lemma *compMemTypes_trns:*
assumes *"CompMemType mem len' arr' tp2 p' stl1"*
and *"CompMemType mem len arr (MTArray len' arr') p'' p'"*
shows *"CompMemType mem len arr tp2 p'' stl1"* ⟨*proof*⟩

lemma *Not_Sub_More_Specific_more_speific:*
assumes *"¬ TypedMemSubPref destl' destl (MTArray l t)"*
shows *"¬ (x<l ∧ TypedMemSubPref destl' (hash destl (ShowL$_{nat}$ x)) t)"* ⟨*proof*⟩

lemma *limitedMemoryChange:*
assumes *"¬TypedMemSubPrefPtrs mem x t p'' prnt"*
and *"p'' ≠ prnt"*
and *"∀ locs. locs ≠ (hash prnt (ShowL$_{nat}$ i)) ⟶ accessStore locs mem = accessStore locs mem'"*
and *"∃ p. accessStore (hash prnt (ShowL$_{nat}$ i)) mem' = Some (MPointer p)"*
and *"MCon (MTArray x t) mem p''"*
shows *"MCon (MTArray x t) mem' p''"* ⟨*proof*⟩

lemma *mconCopySingle:*
assumes *"∀ locs. ¬ LSubPrefL2 locs l ⟶ accessStore locs cd' = accessStore locs c''"*
and*"∀ loc. LSubPrefL2 loc l ⟶ accessStore loc cd' = None"*
and *"MCon struct cd' x2"*
shows *"MCon struct c'' x2"* ⟨*proof*⟩

lemma *mcon_accessStore:*
assumes *"MCon (MTArray len arr) cd loc"*
and *"len > 0"*
and *"i<len"*
obtains *(1) loc2 len' arr'*
where *"accessStore (hash loc (ShowL$_{nat}$ i)) cd = Some (MPointer loc2)"*
and *"arr = MTArray len' arr'"*
| (2) val tp
where *"accessStore (hash loc (ShowL$_{nat}$ i)) cd = Some (MValue val)"*
and *"arr = MTValue tp"*
⟨*proof*⟩

lemma *SameMCon_imps_MConAccessSame:*
assumes *"MCon t m1 l1"*
and *"MCon t m2 l2"*
shows *"MConAccessSame l1 l2 t m1 m2"* ⟨*proof*⟩

lemma *MCon_imps_TypedMemSubPref_Some:*
assumes *"TypedMemSubPref x' loc t"*
and *"MCon t m loc"*
and *"∀ l l'. TypedMemSubPref l loc t ∧ accessStore l m = Some (MPointer l') ⟶ l' = l"*
shows *"∃ v. accessStore x' m = Some v"*
⟨*proof*⟩

lemma *mcon_typedptrs_imps_existence:*
assumes *"MCon (MTArray x t) m1 l1"*
shows *"∀ l1'. TypedMemSubPrefPtrs m1 x t l1 l1' ⟶ (∃ t. MCon t m1 l1')"* ⟨*proof*⟩

lemma *memSet_selfPoint:*

assumes *"MCon t' m tL"*
and *"x'' ∈ GetAllMemoryLocations t' tL m"*
and *"∀ l l'. TypedMemSubPref l tL t' ∧ accessStore l m = Some (MPointer l') ⟶ l' = l"*
and *"accessStore x'' m = Some (MPointer locs')"*
shows *"locs' = x''"* ⟨*proof*⟩

lemma *moreSpecificTypedSubPref:*
assumes *"(∀ l l'. TypedMemSubPref l destl (MTArray x t) ∧ accessStore l v' = Some (MPointer l') ⟶ l' = l)"*
shows *"∀ i<x. (∀ l l'. TypedMemSubPref l (hash destl (ShowL$_{nat}$ i)) t ∧ accessStore l v' = Some (MPointer l') ⟶ l' = l)"* ⟨*proof*⟩

lemma *MCon_mem_preserved_disjoint_update:*
assumes *"(¬LSubPrefL2 destl' destl) ∧ ¬LSubPrefL2 destl destl' "*
shows *"(∀ l l'. TypedMemSubPref l destl' t' ∧ accessStore l v' = Some (MPointer l') ⟶ l' = l)*
∧ (∀ destl'. ¬ LSubPrefL2 destl' destl ⟶ accessStore destl' v' = accessStore destl' v'')
∧ MCon t' v' destl' ⟶ MCon t' v'' destl'" ⟨*proof*⟩

5.4 Typed sub-prefixes

lemma *mtype_size_strict_decrease_array: "mtype_size arr < mtype_size (MTArray len arr)"*
⟨*proof*⟩

lemma *neg_MemLSubPrefL2_imps_TypedMemSubPref:*
assumes *"¬LSubPrefL2 l1 l2"*
shows *"¬TypedMemSubPref l1 l2 t"* ⟨*proof*⟩

abbreviation *mymemory::memoryT*
where *"mymemory ≡*
(|Mapping = fmap_of_list
[(STR ''1.1.0'', MValue STR ''False''),
(STR ''0.1.0'', MValue STR ''True''),
(STR ''1.0'', MPointer STR ''1.0''),
(STR ''1.0.0'', MValue STR ''False''),
(STR ''0.0.0'', MValue STR ''True''),
(STR ''0.0'', MPointer STR ''0.0'')],
Toploc = 1,
Typed_Mapping = fmap_of_list
[(STR ''1.1.0'', MTValue TBool),
(STR ''0.1.0'', MTValue TBool),
(STR ''1.0'', MTArray 2 (MTValue TBool)),
(STR ''1.0.0'', MTValue TBool),
(STR ''0.0.0'', MTValue TBool),
(STR ''0.0'', MTArray 2 (MTValue TBool))]|)"

lemma *reversible_CompMemJustType_imps_same:*
assumes *"CompMemJustType t1 t2"*
and *"CompMemJustType t2 t1"*
shows *"t1 = t2"*
⟨*proof*⟩

lemma *CompMemTypeSameLocsSameType:*
assumes *"MCon (MTArray len arr) mem p'"*
and *"CompMemType mem len arr tp1 p' stl1"*
and *"CompMemType mem len arr tp2 p' stl1"*
shows *"tp2 = tp1"*
⟨*proof*⟩

lemma *CompMemType_asc_withSharedTarget:*
assumes *"CompMemType mem len x (MTArray x' t') p' stl2"*
and *"CompMemType mem len x tp1 p' stl1"*

and *"MCon (MTArray len x) mem p'"*
and *"TypedMemSubPrefPtrs mem x' t' stl2 stl1"*
shows *"CompMemType mem x' t' tp1 stl2 stl1"* ⟨*proof*⟩

lemma *TypedMemSubPrefOneWay:*
assumes *"TypedMemSubPrefPtrs mem x1 t1 stl1 stl2"*
and *"CompMemJustType (MTArray x1 t1) (MTArray x2 t2)"*
and *"MCon (MTArray x1 t1) mem stl1"*
and *"MCon (MTArray x2 t2) mem stl2"*
and *"CompMemType mem x1 t1 (MTArray x2 t2) stl1 stl2"*
shows *"¬TypedMemSubPrefPtrs mem x2 t2 stl2 stl1"*
⟨*proof*⟩

lemma *TypedMemSubPrefPtrs_imps_notsame:*
assumes *"TypedMemSubPrefPtrs m len'' arr'' stl1 prnt"*
and *"MCon (MTArray len'' arr'') m stl1"*
shows *"stl1 ≠ prnt"* ⟨*proof*⟩

lemma *existingLocation_imps_allLocs:*
assumes *"CompMemType m len (MTArray len' arr') subT prnt stl1"*
and *"MCon (MTArray len (MTArray len' arr')) m prnt"*
shows *"∀i<len. ∀l. accessStore (hash prnt (ShowL$_{nat}$ i)) m = Some (MPointer l) ∧ l≠stl1 ∧ TypedMemSubPrefPtrs m len' arr' l stl1 ⟶ CompMemType m len' arr' subT l stl1"*
⟨*proof*⟩

lemma *existingLocation_imps_allLocs_same:*
assumes *"CompMemType m len t subT prnt stl1"*
and *"MCon (MTArray len t) m prnt"*
shows *"∀i<len. ∀l. accessStore (hash prnt (ShowL$_{nat}$ i)) m = Some (MPointer l) ∧ l =stl1 ⟶ subT = t"*
⟨*proof*⟩

lemma *CompMemType_imps_in_GetAllMemoryLocations_ptr:*
assumes *"MCon (MTArray len2 arr2) mem stl2"*
and *"CompMemType mem len2 arr2 (MTArray len3 arr3) stl2 locs"*
shows *"∀i<len3. hash locs (ShowL$_{nat}$ i) ∈ GetAllMemoryLocations (MTArray len2 arr2) stl2 mem"* ⟨*proof*⟩

lemma *typedPrefix_imp_SubPref:*
shows *"TypedMemSubPref child parent t ⟶ LSubPrefL2 child parent"*
⟨*proof*⟩

Locations inside element `i < x` of an array are never below index `x`.

lemma *TypedMemSubPref_not_below_index:*
assumes *"TypedMemSubPref l p (MTArray x t)"*
and *"LSubPrefL2 l' l"*
shows *"¬ LSubPrefL2 l' (hash p (ShowL$_{nat}$ x))"*
⟨*proof*⟩

corollary *TypedMemSubPref_index_disjoint:*
"TypedMemSubPref l p (MTArray x t) ⟹ ¬ LSubPrefL2 l (hash p (ShowL$_{nat}$ x))"
⟨*proof*⟩

corollary *TypedMemSubPref_index_disjoint_typed:*
"TypedMemSubPref l p (MTArray x t) ⟹ ¬ TypedMemSubPref l (hash p (ShowL$_{nat}$ x)) t'"
⟨*proof*⟩

corollary *TypedMemSubPref_hash_not_below_index:*
"TypedMemSubPref l p (MTArray x t) ⟹ ¬ LSubPrefL2 (hash l k) (hash p (ShowL$_{nat}$ x))"
⟨*proof*⟩

corollary *TypedMemSubPref_hash_index_disjoint_typed:*

"TypedMemSubPref l p (MTArray x t) ⟹ ¬ TypedMemSubPref (hash l k) (hash p (ShowL$_{nat}$ x)) t'"
⟨proof⟩

lemma CompMemTypeSubTypes_neg:
shows "¬CompMemType v' x1 t (MTArray x1 t) l destl'"
⟨proof⟩

lemma CompMemType_imps_Mid:
assumes "CompMemType mem len arr (MTArray x11 x12) pParentPtr p"
shows "arr = MTArray x11 x12 ∧ (∃i<len. accessStore (hash pParentPtr (ShowL$_{nat}$ i)) mem = Some (MPointer p))
∨ (∃midP subL subA i.
CompMemType mem len arr (MTArray subL subA) pParentPtr midP
∧ accessStore (hash midP (ShowL$_{nat}$ i)) mem = Some (MPointer p)
∧ i < subL
∧ subA = MTArray x11 x12)"
⟨proof⟩

5.5 Locations above the top location

lemma AllPtrsNotTop_MCon:
assumes "lessThanTopLocs mem"
shows "∀tl. tl ≥ Toploc mem ∧ MCon t mem ptr⟶
(¬LSubPrefL2 ptr (ShowL$_{nat}$ tl))∧
(¬LSubPrefL2 (ShowL$_{nat}$ tl) ptr)"
⟨proof⟩

lemma lessThanTop_topChange_MCon:
assumes "∀locs. ¬ LSubPrefL2 locs (ShowL$_{nat}$ (Toploc mo)) ∨ locs = (ShowL$_{nat}$ (Toploc mo)) ⟶ accessStore locs mo = accessStore locs m"
and "MCon tp mo x2"
and "¬ LSubPrefL2 (ShowL$_{nat}$ (Toploc mo)) x2 ∧ ¬ LSubPrefL2 x2 (ShowL$_{nat}$ (Toploc mo))"
and "lessThanTopLocs mo"
shows " MCon tp (m) x2" ⟨proof⟩

lemma MCon_memory_transfer:
assumes "MCon t mem_old ptr_loc"
and "¬ LSubPrefL2 ptr_loc toploc_new ∧ ¬ LSubPrefL2 toploc_new ptr_loc"
and "∀len arr loc. t = MTArray len arr ∧ TypedMemSubPrefPtrs mem_old len arr ptr_loc loc
⟶ ¬ LSubPrefL2 loc toploc_new ∧ ¬ LSubPrefL2 toploc_new loc"
and "∀loc. ¬ LSubPrefL2 loc toploc_new ∨ loc = toploc_new
⟶ accessStore loc mem_new = accessStore loc mem_old"
shows "MCon t mem_new ptr_loc"
⟨proof⟩

5.6 Preservation of type relations by copying

lemma cpm2mCompMemTypeOld_imps_CompMemType:
assumes " CompMemType mem' len x tp2 stl1 stl2"
and "∀locs. ¬ LSubPrefL2 locs l ⟶ accessStore locs mem' = accessStore locs m'"
and "¬ LSubPrefL2 stl1 l"
and "∀loc. LSubPrefL2 loc l ⟶ accessStore loc mem' = None"
and "MCon (MTArray len x) mem' stl1"
shows " CompMemType m' len x tp2 stl1 stl2" ⟨proof⟩

lemma inv_cpm2mTPrefOld_imps_TPref:
assumes "¬TypedMemSubPrefPtrs mem' len x stl1 stl2"
and "∀locs. ¬ LSubPrefL2 locs l ⟶ accessStore locs mem' = accessStore locs m'"
and "¬ LSubPrefL2 stl1 l"
and "∀loc. LSubPrefL2 loc l ⟶ accessStore loc mem' = None"

and *"MCon (MTArray len x) mem' stl1"*
shows *"¬TypedMemSubPrefPtrs m' len x stl1 stl2"* ⟨*proof*⟩

lemma *selfPoint_imps_TypedMemSubPref:*
assumes *"∀ l l'. TypedMemSubPref l ld (MTArray x t) ∧ accessStore l v' = Some (MPointer l') ⟶ l' = l"*
and *"TypedMemSubPrefPtrs v' x t ld x3"*
shows *"TypedMemSubPref x3 ld (MTArray x t)"* ⟨*proof*⟩

lemma *CompMemType_preservation_induction:*
assumes *store_preserved: "∀ locs. ¬ LSubPrefL2 locs hash_loc ⟶ accessStore locs v' = accessStore locs v''"*
and *no_prefix_conflict: "∀ l. TypedMemSubPref l destl (MTArray x t) ⟶ ¬ LSubPrefL2 l hash_loc"*
and *store_consistency: "∀ l. TypedMemSubPref l destl (MTArray x t) ⟶ accessStore l v' = accessStore l v''"*
and *type_exists: "CompMemType v' x t (MTArray x11 x12) destl destl' ∧ (∀ i<x11. accessTypeStore (hash destl' (ShowL$_{nat}$ i)) v' = Some x12)"*
and *self_pointers: "∀ l l'. TypedMemSubPref l destl (MTArray x t) ∧ accessStore l v' = Some (MPointer l') ⟶ l' = l"*
and *type_preserved: "∀ i<x11. accessTypeStore (hash destl' (ShowL$_{nat}$ i)) v' = accessTypeStore (hash destl' (ShowL$_{nat}$ i)) v''"*
shows *"CompMemType v'' x t (MTArray x11 x12) destl destl' ∧ (∀ i<x11. accessTypeStore (hash destl' (ShowL$_{nat}$ i)) v'' = Some x12)"*
⟨*proof*⟩

lemma *CompMemTypeValue_preservation_induction:*
assumes *store_preserved: "∀ locs. ¬ LSubPrefL2 locs hash_loc ⟶ accessStore locs v' = accessStore locs v''"*
and *no_prefix_conflict: "∀ l. TypedMemSubPref l destl (MTArray x t) ⟶ ¬ LSubPrefL2 l hash_loc"*
and *store_consistency: "∀ l. TypedMemSubPref l destl (MTArray x t) ⟶ accessStore l v' = accessStore l v''"*
and *type_exists: "accessTypeStore destl' v' = Some (MTValue ttt) ∧ CompMemType v' x t (MTValue ttt) destl destl'"*
and *self_pointers: "∀ l l'. TypedMemSubPref l destl (MTArray x t) ∧ accessStore l v' = Some (MPointer l') ⟶ l' = l"*
and *type_preserved: "accessTypeStore destl' v' = accessTypeStore destl' v''"*
shows *"CompMemType v'' x t (MTValue ttt) destl destl' ∧ accessTypeStore destl' v'' = Some (MTValue ttt)"*
⟨*proof*⟩

lemma *CompMemType_extend:*
assumes *comp_exists: "∃ t''. CompMemType v'' x t t'' destl destl' ∧ accessTypeStore destl' v'' = Some t''"*
shows *"∃ t''. CompMemType v'' (Suc x) t t'' destl destl' ∧ accessTypeStore destl' v'' = Some t''"*
⟨*proof*⟩

lemma *CompMemType_extend_array:*
assumes *comp_exists: "CompMemType v'' x t (MTArray x11' x12') destl destl'*
∧ (∀ i<x11'. accessTypeStore (hash destl' (ShowL$_{nat}$ i)) v'' = Some x12')"
shows *"CompMemType v'' (Suc x) t (MTArray x11' x12') destl destl'*
∧ (∀ i<x11'. accessTypeStore (hash destl' (ShowL$_{nat}$ i)) v'' = Some x12')"
⟨*proof*⟩

end

6 Type consistency for storage (TypeSafe_Storage)

theory *TypeSafe_Storage*
imports *TypeSafe_Base_Types*
begin

Size measure on storage types, used to rule out impossible recursion branches.

```
fun stype_size :: "stypes ⇒ nat" where
  "stype_size (STValue _) = 1"
| "stype_size (STArray _ S) = 1 + stype_size S"
| "stype_size (STMap _ S) = 1 + stype_size S"
```

6.1 Storage type conformity

The SCon function works similarly to the MCon function and searches a given storage starting at location (loc) to ensure all sub locations conform to their given types. There are two main differences to MCon. 1. The support for the STMap type. 2. There is no need to look up pointer locations when indexing through arrays, storage does not support pointers in solidity and so a direct lookup can be performed.

```
primrec SCon :: "stypes ⇒ location ⇒ storageT ⇒bool"
  where
    "SCon (STValue typ) loc fm= ((typeCon typ (accessStorage typ loc fm)))"
  |"SCon (STArray len arr) loc fm = (∀ i<len. SCon arr (hash loc (ShowL_nat i)) fm)"
  |"SCon (STMap fromTyp toTyp) loc fm = (∀ i::String.literal. (typeCon fromTyp i) ⟶ SCon toTyp (hash
loc i) fm)"


lemma subSCon:
  assumes "SCon (STArray (Suc x) t') srcl srcMem"
  shows "SCon (STArray (x) t') srcl srcMem" ⟨proof⟩


primrec TypedStoSubpref :: "location ⇒ location ⇒ stypes ⇒ bool"
  where
    "TypedStoSubpref child parent (STValue typ) = (child = parent)"
  | "TypedStoSubpref child parent (STArray len arr) = (child = parent ∨
                                                       (∃ i<len. TypedStoSubpref child (hash parent
(ShowL_nat i)) arr))"
  | "TypedStoSubpref child parent (STMap fromTyp toTyp) = ((child = parent)
                                ∨ (∃ i::String.literal. (typeCon fromTyp i) ∧ (TypedStoSubpref child
(hash parent i) toTyp)))"


lemma TypedStoSubpref_b:
  assumes "TypedStoSubpref l1 l2 t"
  shows "(∃ a. hash l2 a = l1) ∨ l1 = l2" ⟨proof⟩


lemma TypedStoSubpref_imp_LSubPrefL2:
  shows "TypedStoSubpref child parent t ⟶ LSubPrefL2 child parent"
  ⟨proof⟩

lemma neg_LSubPrefL2_imps_neg_TypedStoSubpref:
  assumes "¬ LSubPrefL2 child prnt"
  shows "¬ TypedStoSubpref child prnt t"
  ⟨proof⟩

lemma stoMoreSpecificTypedSubpref:
  assumes "∀ l l'. TypedStoSubpref l destl (STArray x t) ∧ accessStore l v' = Some (MPointer l') ⟶ l'
= l"
  shows "∀ i<x. ∀ l l'. TypedStoSubpref l (hash destl (ShowL_nat i)) t ∧ accessStore l v' = Some
(MPointer l') ⟶ l' = l"
⟨proof⟩


lemma TypedStoSubpref_hashes:
  shows "TypedStoSubpref l2' (hash loc kv) t ⟶ l2' ≠ loc" ⟨proof⟩

lemma TypedStoSubpref_sameLoc:
  shows "TypedStoSubpref l1 l1 tp"
```

⟨*proof*⟩

lemma *typedStoSub_imps_negInv:*
assumes *"short ≠ short'"*
shows *"TypedStoSubpref short short' t' ⟶ ¬ TypedStoSubpref short' short t"*
⟨*proof*⟩

primrec *subStoTp :: "stypes ⇒ stypes ⇒ bool"*
where
"subStoTp child (STValue typ) = (child = (STValue typ))"
| "subStoTp child (STArray len arr) = (child = (STArray len arr) ∨ (subStoTp child arr))"
| "subStoTp child (STMap fromTyp toTyp) = ((child = (STMap fromTyp toTyp)) ∨ (subStoTp child toTyp))"

primrec *CompStoType :: "stypes ⇒ stypes ⇒ location ⇒ location ⇒ bool"*
where
"CompStoType (STValue typ) childtp parentloc childloc= (childtp = (STValue typ) ∧ (parentloc = childloc))"
| "CompStoType (STArray len arr) childtp parentloc childloc = (((childtp = STArray len arr) ∧ parentloc = childloc) ∨
(∃ i<len. CompStoType arr childtp (hash parentloc (ShowL$_{nat}$ i)) childloc))"
| "CompStoType (STMap fromTyp toTyp) childtp parentloc childloc = ((childtp = STMap fromTyp toTyp ∧ parentloc = childloc) ∨ (∃ i::String.literal.
(typeCon fromTyp i ∧ (CompStoType toTyp childtp (hash parentloc i) childloc))))"

lemma *CompStoType_imps_TypedStoSubpref:*
assumes *"CompStoType t tp l'' l"*
shows *"TypedStoSubpref l l'' t"* ⟨*proof*⟩

lemma *CompStoType_sameLocNdTyp:*
shows *"CompStoType tp1 tp1 l1 l1"*
⟨*proof*⟩

lemma *CompStoType_imp_subType:*
shows *"loc ≠ loc' ∧ CompStoType t1 t2 loc loc' ⟶ (∃ t. subStoTp t1 t ∧ t = t1)"*
⟨*proof*⟩

lemma *CompStoType_imps_subloc:*
shows *"loc ≠ l2' ∧ CompStoType t1 t2 loc l2' ⟶ (∃ i. hash loc i = l2')"*
⟨*proof*⟩

lemma *CompStoType_sameLoc_sameType:*
assumes *"CompStoType t1 t2 l1 l1"*
shows *"t1 = t2"*
⟨*proof*⟩

lemma *comp_stotype_size_le:*
"CompStoType t1 t2 l1 l2 ⟹ stype_size t2 ≤ stype_size t1"
⟨*proof*⟩

lemma *comp_stotype_same_type_same_loc:*
shows *"CompStoType t t l1 l2 ⟹ l1 = l2"*
⟨*proof*⟩

definition *compPointers :: "stack ⇒ (String.literal, type × denvalue) fmap ⇒ bool"* **where**
"compPointers st denval = (∀ tp1 tp2 l1 l2 l1' l2' stl1 stl2.
(type.Storage tp1, l1) |∈| fmran denval ∧ (type.Storage tp2, l2) |∈| fmran denval ∧
((l1 = Stackloc l1' ∧ accessStore l1' st = Some(KStoptr stl1)) ∨ (l1 = Storeloc stl1)) ∧
((l2 = Stackloc l2' ∧ accessStore l2' st = Some(KStoptr stl2)) ∨ (l2 = Storeloc stl2)) ⟶

```
  (if TypedStoSubpref stl2 stl1 tp1 then CompStoType tp1 tp2 stl1 stl2
   else if TypedStoSubpref stl1 stl2 tp2 then CompStoType tp2 tp1 stl2 stl1
   else True))"
```

Overwriting one stack slot with a memory pointer preserves *compPointers*.

lemma *compPointers_stack_update:*
assumes *"compPointers sk den"*
and *"∀loc. loc ≠ l ⟶ accessStore loc sk = accessStore loc sk'"*
and *"accessStore l sk' = Some (KMemptr p)"*
shows *"compPointers sk' den"*
⟨*proof*⟩

The pointer-compatibility condition of *compPointers* is symmetric.

lemma *compPointers_if_sym:*
assumes *"if TypedStoSubpref l1 l2 t2 then CompStoType t2 t1 l2 l1*
else if TypedStoSubpref l2 l1 t1 then CompStoType t1 t2 l1 l2 else True"
shows *"if TypedStoSubpref l2 l1 t1 then CompStoType t1 t2 l1 l2*
else if TypedStoSubpref l1 l2 t2 then CompStoType t2 t1 l2 l1 else True"
⟨*proof*⟩

lemma *TypedStoSubpref_child_imps_prnt:*
assumes *"TypedStoSubpref (hash x4 i) p' t"*
and *"CHR ''.'' ∉ set(String.explode i)"*
and *"hash x4 i ≠ p'"*
shows *"TypedStoSubpref x4 p' t"*
⟨*proof*⟩

lemma *TypedStoSubpref_specific_unreachanble_arry:*
assumes *"destl ≠ srcl"*
and *"¬ TypedStoSubpref srcl destl (STArray x t')"*
and *"i < x"*
shows *"¬ TypedStoSubpref (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t'"*
⟨*proof*⟩

lemma *NotRelatedPrnt_imps_notRelatedChild:*
assumes *"¬ TypedStoSubpref x4 l''' t"*
and *"¬ TypedStoSubpref l''' x4 struct"*
and *"TypedStoSubpref l l''' t"*
shows *"¬ TypedStoSubpref l x4 struct "*
⟨*proof*⟩

lemma *CompStoType_same_type_same_depth:*
assumes *"CompStoType tp'' t prnt l1"*
and *"CompStoType tp'' t prnt' l2"*
and *"location_depth prnt = location_depth prnt'"*
shows *"location_depth l1 = location_depth l2"* ⟨*proof*⟩

lemma *TypedStoSubpref_imp_same_depth:*
assumes *"TypedStoSubpref l pa tp''"*
and *"l ≠ pa"*
shows *"∃a. location_depth (hash pa a) = location_depth l"*
⟨*proof*⟩

lemma *same_depth_imp_neg_TypedStoSubpref:*
assumes *"location_depth l = location_depth pa"*
and *"l ≠ pa"*
shows *"¬TypedStoSubpref l pa tp''"* ⟨*proof*⟩

lemma *CompStoType_unique_location:*
assumes *"CompStoType struct tp1 (hash loc suffix1) p'"*
and *"CompStoType struct tp2 (hash loc suffix2) p'"*

```
    and "suffix1 ≠ suffix2"
    and "CHR ''.'' ∉ set(String.explode suffix1)"
    and "CHR ''.'' ∉ set(String.explode suffix2)"
  shows "False"
⟨proof⟩

lemma sublocs_nonchanged_SCon:
  assumes "¬ TypedStoSubpref p' x4 struct"
    and "∀ locs t''.
       locs ≠ p' ∧ ¬ TypedStoSubpref locs p' (STArray x t''') ⟶
       accessStorage t'' locs sto = accessStorage t'' locs sto'"
    and "SCon struct x4 sto"
    and "¬ TypedStoSubpref x4 p' (STArray x t''')"
  shows "SCon struct x4 sto'" ⟨proof⟩


lemma singleLocChanged_nonchanged_SCon:
  assumes "∀ l. l ≠ locationChanged ⟶ sto' $$ l = sto $$ l"
    and "x4 ≠ locationChanged"
    and "SCon struct x4 sto"
    and "¬ TypedStoSubpref l'' x4 struct"
    and "¬ TypedStoSubpref x4 l'' t''"
    and " TypedStoSubpref locationChanged l'' t'' ∧ CompStoType t'' (STValue t') l'' locationChanged"
  shows " SCon struct x4 sto'" ⟨proof⟩

lemma SCon_imps_sublocs:
  assumes "CompStoType t' struct l x2"
    and "SCon t' l sto'"
  shows "SCon struct x2 sto'"
  ⟨proof⟩


lemma TypedStoSubpref_shared_parent_related:
  assumes "TypedStoSubpref lc l'' t''"
    and "TypedStoSubpref lc x4 struct"
  shows "∃ pref. hash l'' pref = x4 ∨ hash x4 pref = l'' ∨ l'' =x4 "
⟨proof⟩


lemma subLocs:
  assumes "SCon struct x4 sto"
    and "∀ l struct. ¬TypedStoSubpref locationChanged l struct ⟶ sto' $$ l = sto $$ l"
    and "(∀ i'. (hash x4 i') ≠ locationChanged)"
    and "x4 ≠ locationChanged"
  shows " SCon struct x4 sto'" ⟨proof⟩


lemma stvalueLocationsInduct:
  assumes "CompStoType struct (STValue t') x4 locationChanged"
    and " SCon struct x4 sto"
    and "SCon (STValue t') locationChanged sto'"
    and "hash x4 i'' = locationChanged"
    and "∀ l struct. ¬TypedStoSubpref locationChanged l struct ⟶ sto' $$ l = sto $$ l"
    and "∀ l. l ≠ locationChanged ⟶ sto' $$ l = sto $$ l"
  shows "SCon struct x4 sto'" ⟨proof⟩

lemma SCon_value_write_imps_SCon:
  assumes comp: "CompStoType struct (STValue t') x4 locationChanged"
    and scOld: "SCon struct x4 sto"
    and scNew: "SCon (STValue t') locationChanged sto'"
    and nonLocChanged: "∀ l. l ≠ locationChanged ⟶ sto' $$ l = sto $$ l"
  shows "SCon struct x4 sto'"
⟨proof⟩
```

lemma *CompStoType_subloc_type_transfer:*
assumes *"CompStoType t'' (STValue t') l'' locationChanged "*
and *" CompStoType t'' struct l'' x4 "*
and *"TypedStoSubpref locationChanged x4 struct"*
shows *" CompStoType struct (STValue t') x4 locationChanged"* ⟨*proof*⟩

lemma *CompStoType_sameLocs_sameType:*
assumes *"CompStoType t t' l''' x4"*
and *"CompStoType t struct l''' x4"*
shows *"struct = t'"* ⟨*proof*⟩

lemma *CompStoType_trns:*
assumes *"CompStoType struct t'' x4 l'''"*
and *"CompStoType t'' t' l'' locationChanged"*
shows *"CompStoType struct t' x4 locationChanged"* ⟨*proof*⟩

lemma *NotReachablePrnt_imps_notReachableChild:*
assumes *"¬ TypedStoSubpref x4 l''' t"*
and *"¬ TypedStoSubpref l''' x4 struct"*
and *"CompStoType t t' l''' l"*
shows *"¬ TypedStoSubpref x4 l t'"*
⟨*proof*⟩

lemma *SCon_noDots:*
assumes *"SCon tp (hash destl i) v'"*
and *"∀ i' t. accessStorage t (hash (hash destl i) i') v' = accessStorage t (hash (hash destl i) i') v''"*
and *" ∀ t. accessStorage t (hash destl i) v' = accessStorage t (hash destl i) v''"*
and *"CHR ''.'' ∉ set(String.explode i)"*
shows *"SCon tp (hash destl i) v''"* ⟨*proof*⟩

lemma *Sto_divergence_imps_notsubloc:*
shows *"¬TypedStoSubpref (hash destl (ShowL$_{nat}$ x)) destl (STArray x tp)"*
⟨*proof*⟩

lemma *NoSubChanged_SCon_Preserved:*
assumes *"SCon t' destl srcMem"*
and *"∀ i' t''. accessStorage t'' (hash destl i') srcMem = accessStorage t'' (hash destl i') v'"*
and *"∀ t''. accessStorage t'' destl srcMem = accessStorage t'' destl v'"*
shows *"SCon t' destl v'"* ⟨*proof*⟩

lemma *SCon_NoChange:*
assumes *"∀ destl'. TypedStoSubpref destl' destl t' ⟶ (∀ t. accessStorage t destl' srcMem = accessStorage t destl' v')"*
and *"SCon t' destl srcMem"*
shows *"SCon t' destl v'"* ⟨*proof*⟩

lemma *TypedStoSubpref_arr_longer:*
assumes *"(∀ subL. TypedStoSubpref subL srcl (STArray (Suc x) t') ⟶ ¬ TypedStoSubpref subL destl (STArray (Suc x) t'))"*
shows *"(∀ subL. TypedStoSubpref subL srcl (STArray x t') ⟶ ¬ TypedStoSubpref subL destl (STArray x t'))"*
⟨*proof*⟩

lemma *SCon_preserved_disjoint_change:*
assumes *"∀ destl'. ¬ TypedStoSubpref destl' destl t' ⟶ (∀ t. accessStorage t destl' srcMem = accessStorage t destl' v')"*
and *" ∀ subL. TypedStoSubpref subL srcl t' ⟶ ¬ TypedStoSubpref subL destl t'"*
and *" SCon t' srcl srcMem"*
shows *"SCon t' srcl v'"* ⟨*proof*⟩

lemma *CompStoType_sharedSub:*
assumes *"CompStoType pParentT tp1 pParentPtr p"*
and *"TypedStoSubpref p stl2 tp2"*
and *"CompStoType pParentT tp2 pParentPtr stl2"*
shows *"CompStoType tp2 tp1 stl2 p "* ⟨*proof*⟩

lemma *SCon_sub_imps_Parent:*
assumes *" CompStoType struct (STArray x t''') x4 p'"*
and *"SCon (STArray x t''') p' sto'"*
and *"SCon struct x4 sto"*
and *"t' = STArray x t''' "*
and *"∀ locs t''. locs ≠ p' ∧ ¬ TypedStoSubpref locs p' (STArray x t''') ⟶ accessStorage t'' locs sto = accessStorage t'' locs sto'"*
shows *"SCon struct x4 sto'"* ⟨*proof*⟩

lemma *SCon_update_array_subloc_cases:*
assumes *cmp:*
"if TypedStoSubpref p' x4 struct then CompStoType struct t' x4 p' else if TypedStoSubpref x4 p' t' then CompStoType t' struct p' x4 else True"
and *scParent: "SCon (STArray x t''') p' sto'"*
and *scChanged: "SCon t' p' sto'"*
and *scOld: "SCon struct x4 sto"*
and *tDef: "t' = STArray x t'''"*
and *unchanged:*
"∀ locs t''. locs ≠ p' ∧ ¬ TypedStoSubpref locs p' (STArray x t''') ⟶ accessStorage t'' locs sto = accessStorage t'' locs sto'"
shows *"SCon struct x4 sto'"*
⟨*proof*⟩

Updating the storage array at `l` preserves *SCon* at a location *x4* that shares a parent `lr` with `l`.

lemma *SCon_preserved_array_update_related:*
assumes *rel: "CompStoType tr (STArray x t) lr l"*
and *cmp: "CompStoType tr struct lr x4"*
and *new: "SCon (STArray x t) l sto'"*
and *old: "SCon struct x4 sto"*
and *same: "∀ locs t''. locs ≠ l ∧ ¬ TypedStoSubpref locs l (STArray x t) ⟶ accessStorage t'' locs sto = accessStorage t'' locs sto'"*
shows *"SCon struct x4 sto'"*
⟨*proof*⟩

The same holds when *x4* and the parent `lr` of `l` are unrelated.

lemma *SCon_preserved_array_update_unrelated:*
assumes *rel: "CompStoType tr (STArray x t) lr l"*
and *n1: "¬ TypedStoSubpref x4 lr tr"*
and *n2: "¬ TypedStoSubpref lr x4 struct"*
and *old: "SCon struct x4 sto"*
and *same: "∀ locs t''. locs ≠ l ∧ ¬ TypedStoSubpref locs l (STArray x t) ⟶ accessStorage t'' locs sto = accessStorage t'' locs sto'"*
shows *"SCon struct x4 sto'"*
⟨*proof*⟩

Combined form, taking the pointer-compatibility fact from *compPointers*.

lemma *SCon_preserved_array_update:*
assumes *rel: "CompStoType tr (STArray x t) lr l"*
and *ptr: "if TypedStoSubpref lr x4 struct then CompStoType struct tr x4 lr else if TypedStoSubpref x4 lr tr then CompStoType tr struct lr x4 else True"*
and *n2: "¬ TypedStoSubpref lr x4 struct"*
and *new: "SCon (STArray x t) l sto'"*
and *old: "SCon struct x4 sto"*
and *same: "∀ locs t''. locs ≠ l ∧ ¬ TypedStoSubpref locs l (STArray x t) ⟶ accessStorage t'' locs sto = accessStorage t'' locs sto'"*
shows *"SCon struct x4 sto'"*
⟨*proof*⟩

The pointer-compatibility condition carries over from a location *p* to a sub-location *l1* of *p*.

lemma *compPointers_if_child:*
  assumes *comps: "if TypedStoSubpref p l2 t2 then CompStoType t2 tp l2 p*
else if TypedStoSubpref l2 p tp then CompStoType tp t2 p l2 else True"
    and *child: "CompStoType tp t1 p l1"*
  shows *"if TypedStoSubpref l2 l1 t1 then CompStoType t1 t2 l1 l2*
else if TypedStoSubpref l1 l2 t2 then CompStoType t2 t1 l2 l1 else True"
⟨*proof*⟩

end

7 Type consistency for environments and states (TypeSafe_Def)

theory *TypeSafe_Def*
  imports *TypeSafe_Memory TypeSafe_Storage*

begin

7.1 Ensuring unique locations

No two entries of a *Denvalue* map share a location.

definition *unique_locations :: "(String.literal, type × denvalue) fmap ⇒ bool"* **where**
  "unique_locations denval = (∀x y. x |∈| fmran denval ∧ y |∈| fmran denval ∧ snd x = snd y ⟶ x = y)"

If an environment has unique locations, two entries of its *Denvalue* with the same location are equal. Hence no two variables share a location.

lemma *uniqueLocs:*
  assumes *"unique_locations (Denvalue ev)"*
    and *"x |∈| fmran (Denvalue ev)"*
    and *"y |∈| fmran (Denvalue ev)"*
    and *"snd x = snd y"*
  shows *"x = y"* ⟨*proof*⟩

7.1.1 Pushing to the stack maintains stackLocations

lemma *stackPushToplocSafe:*
  assumes *"lessThanTopLocs k"*
    and *"k' = push p k"*
  shows *"lessThanTopLocs k'"* ⟨*proof*⟩

lemma *lessThanSome_imps_Locs:*
  assumes *"lessThanTopLocs mem"*
    and *"accessStore p' mem = Some y"*
    and *"LSubPrefL2 p' (ShowL$_{nat}$ x)"*
  shows *"x < Toploc mem"*
⟨*proof*⟩

lemma *lessThanSome_imps_Locs_hash:*
  assumes *"lessThanTopLocs mem"*
    and *" accessStore (hash p' (ShowL$_{nat}$ i)) mem = Some y"*
    and *"LSubPrefL2 p' (ShowL$_{nat}$ x)"*
  shows *"x < Toploc mem"*
⟨*proof*⟩

lemma *subPtrs_nonTop:*
  assumes *"MCon (MTArray x11' x12') mem stl1"*
    and *"(∀tloc loc. Toploc mem ≤ tloc ∧ LSubPrefL2 loc (ShowL$_{nat}$ tloc) ⟶ accessStore loc mem = None)"*
    and *"(∀loc y. accessStore loc mem = Some y ⟶ (∃tloc<Toploc mem. LSubPrefL2 loc (ShowL$_{nat}$ tloc)))"*
    and *"∀locs. ¬ LSubPrefL2 locs (ShowL$_{nat}$ (Toploc mem)) ⟶ accessStore locs mem' = accessStore locs mem"*

```
    and "TypedMemSubPrefPtrs mem' x11' x12' stl1 dloc1"
  shows "¬ LSubPrefL2 dloc1 (ShowL_nat (Toploc mem))"
⟨proof⟩

lemma PreExistMconNotChangeByToploc:
  assumes "(∀ loc y. accessStore loc mem = Some y ⟶ (∃ tloc<Toploc mem. LSubPrefL2 loc (ShowL_nat
tloc)))"
    and "(∀ tloc loc. Toploc mem ≤ tloc ∧ LSubPrefL2 loc (ShowL_nat tloc) ⟶ accessStore loc mem =
None)"
    and "∀ locs. ¬ LSubPrefL2 locs (ShowL_nat(Toploc mem)) ∨ locs =
        (ShowL_nat(Toploc mem)) ⟶ accessStore locs mem = accessStore locs m"
    and "MCon tp1 mem stl1"
  shows "MCon tp1 m stl1" ⟨proof⟩
```

7.2 Balances and message values conform

```
definition balanceTypes :: "accounts ⇒ bool" where
  "balanceTypes acc = (∀ adv. typeCon (TUInt b256) (Bal (acc adv)))"

definition addressFormat::"address ⇒ bool"
  where "addressFormat cur = (typeCon TAddr cur)"

abbreviation envAddressesWellFormed :: "environment ⇒ bool" where
  "envAddressesWellFormed ev ≡ addressFormat (Address ev) ∧ addressFormat (Sender ev)"

definition svalueTypes :: "valuetype ⇒ bool" where
  "svalueTypes sval = typeCon (TUInt b256) sval"
```

7.3 Compatible pointers

Arguments: parent type, parent location, sub-prefix type, sub-prefix location.

```
primrec Subpref :: "stypes ⇒ location ⇒ stypes ⇒ location ⇒ bool"
  where
    "Subpref (STValue typ) Modulatedparent typ2 subloc = ((Modulatedparent = subloc) ∧ typ2 = (STValue
typ))"
  |"Subpref (STArray len arr) Modulatedparent typ2 subloc = (((STArray len arr) = typ2 ∧
Modulatedparent = subloc)
                                                          ∨ ( ∃i::nat. i < len ∧ Subpref arr (hash
Modulatedparent (ShowL_nat i)) typ2 subloc))"
  |"Subpref (STMap fromTyp toTyp) Modulatedparent typ2 subloc = (((STMap fromTyp toTyp) = typ2 ∧
Modulatedparent = subloc)
                                                              ∨(∃i::String.literal. (typeCon fromTyp i) ∧
Subpref toTyp (hash Modulatedparent i) typ2 subloc))"


lemma "¬Subpref (STArray 3 (STValue TBool)) (STR ''1.5'') (STValue TBool) (STR ''4.1.5'')" ⟨proof⟩
lemma "Subpref (STArray 5 (STValue TBool)) (STR ''1.5'') (STValue TBool) (STR ''1.1.5'')" ⟨proof⟩
lemma "Subpref (STArray 5 (STArray 4 (STArray 3 (STValue TBool)))) (STR ''1.5'') ((STValue TBool))
(STR ''1.1.1.1.5'') " ⟨proof⟩
lemma "Subpref (STArray 5 (STArray 4 (STArray 3 (STValue TBool)))) (STR ''1.5'') ((STArray 3 (STValue
TBool))) (STR ''1.1.1.5'') " ⟨proof⟩
```

The value stored for one variable of the environment conforms to its declared type.

```
abbreviation denvalueCon :: "stack ⇒ memoryT ⇒ storageT ⇒ calldataT ⇒ type ⇒ denvalue ⇒ bool"
  where "denvalueCon sck mem sto cd t l ≡
    (case l of Stackloc loc ⇒
       (case accessStore loc sck of None ⇒ False
        | Some (KValue val) ⇒ (case t of Value typ ⇒ typeCon typ val | _ ⇒ False)
        | Some (KCDptr stloc) ⇒ (case t of Calldata struct ⇒ MCon struct cd stloc | _ ⇒ False)
        | Some (KMemptr stloc) ⇒ (case t of type.Memory struct ⇒ MCon struct mem stloc | _ ⇒ False)
        | Some (KStoptr stloc) ⇒ (case t of type.Storage struct ⇒ SCon struct stloc sto | _ ⇒ False))
     | Storeloc loc ⇒ (case t of type.Storage typ ⇒ SCon typ loc sto | _ ⇒ False))"
```

```
definition typeCompat::"(String.literal, type × denvalue) fmap ⇒ stack ⇒ memoryT ⇒ storageT ⇒
calldataT ⇒ bool"
  where "typeCompat den sck0 mem sto cd = (∀t l. (t, l) |∈| fmran (den) ⟶ denvalueCon sck0 mem sto
cd t l)"
```

7.4 Supplementary lemmas

Collection of lemmas which shorten common proof obligations

Support functions to simplify the typeCon expressions definition

```
fun extractValueType :: "stackvalue ⇒ valuetype" where
  "extractValueType (KValue v) = v"
| "extractValueType (KCDptr v) = v"
| "extractValueType (KMemptr v) = v"
| "extractValueType (KStoptr v) = v"
```

A value of a given type is consistent with calldata, memory, and storage.

```
abbreviation valueCon :: "environment ⇒ calldataT ⇒ state ⇒ stackvalue ⇒ type ⇒ bool"
  where "valueCon ev cd st v t ≡
    (case t of Value typ ⇒ typeCon typ (extractValueType v)
     | Calldata struct ⇒ MCon struct cd (extractValueType v)
     | type.Memory struct ⇒ MCon struct (Memory st) (extractValueType v)
     | type.Storage struct ⇒ SCon struct (extractValueType v) (Storage st (Address ev)))"
```

The result of evaluating an expression is consistent with its type and, for pointers, is related to a variable of the environment: calldata and memory pointers reach the value through a stack variable, storage pointers through a storage variable.

```
abbreviation exprResultCon :: "environment ⇒ calldataT ⇒ state ⇒ stackvalue ⇒ type ⇒ bool"
  where "exprResultCon ev cd st v t ≡
    (case t of Value typ ⇒ typeCon typ (extractValueType v) ∧ (∃xx. v = KValue xx)
     | Calldata struct ⇒ MCon struct cd (extractValueType v) ∧ (∃xx. v = KCDptr xx) ∧
         (∃stloc tp'' p. (Calldata tp'', Stackloc stloc) |∈| fmran (Denvalue ev) ∧
            accessStore stloc (Stack st) = Some (KCDptr p) ∧
            (tp'' = struct ∧ v = KCDptr p ∨ (∃len arr. extractValueType v ≠ p ∧
               tp'' = MTArray len arr ∧ CompMemType cd len arr struct p (extractValueType v))))
     | type.Memory struct ⇒ MCon struct (Memory st) (extractValueType v) ∧ (∃xx. v = KMemptr xx) ∧
         (∃stloc tp'' p. (type.Memory tp'', Stackloc stloc) |∈| fmran (Denvalue ev) ∧
            accessStore stloc (Stack st) = Some (KMemptr p) ∧
            (tp'' = struct ∧ v = KMemptr p ∨ (∃len arr. extractValueType v ≠ p ∧
               tp'' = MTArray len arr ∧ CompMemType (Memory st) len arr struct p (extractValueType
v))))
     | type.Storage struct ⇒ SCon struct (extractValueType v) (Storage st (Address ev)) ∧
         (∃xx. v = KStoptr xx) ∧
         (∃stloc tp''. (type.Storage tp'', Storeloc stloc) |∈| fmran (Denvalue ev) ∧
            (tp'' = struct ∧ v = KStoptr stloc ∨
             extractValueType v ≠ stloc ∧ CompStoType tp'' struct stloc (extractValueType v))))"


locale typesafe_base = statement_with_gas +
  assumes methodVarsNoPref: "∀c ct dud i1 i2 t1 t2. i1 ≠ i2 ∧ ep $$ c = Some(ct,dud) ∧ ct $$ i1 =
Some (Var t1) ∧ ct $$ i2 = Some (Var t2)
                              ⟶ (¬TypedStoSubpref i1 i2 t2) ∧ (¬TypedStoSubpref i2 i1 t1)"

context typesafe_base
begin
```

An environment and state are fully initialised if the contract has an initialised account and all contract variables are bound in the `Denvalue`. These are the first steps when creating a new environment (see statement `NEW`).

```
definition fullyInitialised::"environment ⇒ accounts ⇒ stack ⇒ bool"
  where "fullyInitialised env acc sk = (
(∃c ct dud.
          Type (acc (Address env)) = Some (atype.Contract c) ∧
          Contract env = c ∧
```

```
          ep $$ c = Some (ct, dud) ∧
          (∀ id v. ct $$ id = Some (Var v) ⟷
                 Denvalue env $$ id = Some (type.Storage v, Storeloc id)) ∧
          (∀ id v loc. Denvalue env $$ id = Some (type.Storage v, Storeloc loc) ⟶ id = loc) ∧

          (∀ t l p.
              (type.Storage t, Stackloc l) |∈| fmran (Denvalue env) ∧
              accessStore l sk = Some (KStoptr p) ⟶
              (∃ t' l'. (type.Storage t', Storeloc l') |∈| fmran (Denvalue env) ∧
                       CompStoType t' t l' p)))


)"

lemma test_not_vacuous:
  fixes my_contract :: "String.literal"
  fixes n :: "stypes"
  assumes blueprint_def: "my_blueprint = fmupd (STR ''balance'') (Var n) fmempty"
  assumes ep_def: "ep $$ my_contract = Some (my_blueprint, dud)"

  shows "∃ env acc sk. fullyInitialised env acc sk ∧
                     Denvalue env $$ (STR ''balance'') = Some (type.Storage n, Storeloc (STR
''balance''))"
⟨proof⟩


definition AddressTypes :: "accounts ⇒ bool" where
    "AddressTypes ac =
      (∀ adv. case Type (ac adv) of
                Some (atype.Contract c) ⇒ addressFormat adv ∧ c |∈| fmdom ep
              | _ ⇒ True)"

abbreviation accountsWellTyped :: "accounts ⇒ bool" where
  "accountsWellTyped acc ≡ balanceTypes acc ∧ AddressTypes acc"


definition safeContract :: "accounts ⇒ (address ⇒ storageT) ⇒ bool" where
  "safeContract acc sto =
    (∀ e ct dud i tp.
      Type (acc (Address (e::environment))) = Some (atype.Contract (Contract e)) ∧
      ep $$ Contract e = Some (ct, dud) ∧
      fmlookup ct i = Some (Var tp)
      ⟶ SCon tp i (sto (Address (e::environment))))"
```

`safeContract` depends on the accounts only through their types.

```
lemma safeContract_Type_cong:
  assumes "safeContract acc sto"
    and "∀ a. Type (acc' a) = Type (acc a)"
  shows "safeContract acc' sto"
  ⟨proof⟩
```

7.5 Type consistency for reachable locations

This property ensures that the types stored in the typed mapping of memory are consistent with the types derivable from the denvalue for locations that are reachable from variables in the environment.

```
definition denvalueTypeCorrectness :: "environment ⇒ stack ⇒ memoryT ⇒ bool"
  where
    "denvalueTypeCorrectness env sck mem =
    (∀ t l ptr_loc. (type.Memory t, Stackloc l) |∈| fmran (Denvalue env) ∧
        accessStore l sck = Some (KMemptr ptr_loc) ⟶
        (case t of MTArray len arr ⇒
                (∀ i<len. accessTypeStore (hash ptr_loc (ShowL_nat i)) mem = Some arr)
          | MTValue val ⇒ accessTypeStore ptr_loc mem = Some (MTValue val))
```

```
    )"

definition subPrefixStructuralConsistency :: "memoryT ⇒ bool"
  where
    "subPrefixStructuralConsistency mem =
      (∀ locs tp. accessTypeStore locs mem = Some tp ⟶
          (case accessStore locs mem of Some (MPointer p) ⇒
               (∃ len arr. MCon tp mem p ∧ tp = MTArray len arr
                  ∧ (∀ i<len. accessTypeStore (hash p (ShowL_nat i)) mem = Some arr)
                )
           | Some (MValue v) ⇒
               (∃ val. MCon tp mem locs ∧ tp = MTValue val ∧accessTypeStore locs mem = Some tp)
           | None ⇒ False))"

definition SomeValSomeTyp:: "memoryT ⇒ bool"
  where
    "SomeValSomeTyp mem = (∀ locs. (∃ t. accessStore locs mem = Some t) ⟷ (∃ tt. accessTypeStore locs
mem = Some tt))"
```

7.6 Typesafe environments

TypeSafe states that an environment, accounts, stack, memory, storage, and calldata are mutually type-consistent. It combines two groups of properties:

1. structural properties of the environment and stores, such as unique locations, compatible pointers, and bounded locations;

2. value conformance: for every variable in the *Denvalue*, the string representation of its value conforms to its declared type.

```
definition TypeSafe :: "environment ⇒ accounts ⇒ stack ⇒ memoryT ⇒ (address ⇒ storageT) ⇒
calldataT ⇒ bool"
  where "TypeSafe ev acc sck mem sto cd= (
          typeCompat (Denvalue ev) sck mem (sto (Address ev)) cd ∧
          (unique_locations (Denvalue ev)) ∧
          (compPointers sck (Denvalue ev)) ∧
          (lessThanTopLocs sck) ∧ (lessThanTopLocs mem) ∧ (lessThanTopLocs cd) ∧
          envAddressesWellFormed ev ∧
          accountsWellTyped acc ∧
          (safeContract acc sto) ∧
          (svalueTypes (Svalue ev)) ∧
          denvalueTypeCorrectness ev sck mem ∧
          subPrefixStructuralConsistency mem ∧
          SomeValSomeTyp mem)"
```

Memory of a type-safe state holds values only below its top location.

```
lemma TypeSafe_memory_bounds:
  assumes "TypeSafe ev acc sck mem sto cd"
  shows "∀ loc y. accessStore loc mem = Some y ⟶ (∃ tloc<Toploc mem. LSubPrefL2 loc (ShowL_nat tloc))"
    and "∀ tloc loc. Toploc mem ≤ tloc ∧ LSubPrefL2 loc (ShowL_nat tloc) ⟶ accessStore loc mem =
None"
  ⟨proof⟩
```

7.7 All stack locations in the denvalue have a value in the stack

```
lemma typeSafeAllStacklocsExist:
  assumes "TypeSafe ev acc sck mem sto cd"
  assumes "(t, l2) |∈| fmran (Denvalue ev)"
    and "l2 = Stackloc l"
  shows "∃ v. accessStore l sck = Some v"
⟨proof⟩
```

7.8 *astack* preserves unique locations

lemma *updateEnvUniqueLocs:*
assumes *"TypeSafe ev' (Accounts st) sck' mem' (Storage st) cd'"*
and *"(k',e) = astack ip (v) (p) (sck', ev')"*
shows *"unique_locations (Denvalue e)"* ⟨*proof*⟩

lemma *sameMemTSafe:*
assumes *"TypeSafe ev acc sck mem sto cd"*
shows *"∀struct loc. ((type.Memory struct), (Stackloc loc)) |∈| fmran (Denvalue ev) ∧ accessStore loc sck = Some (KMemptr stloc) ⟶ MCon struct mem stloc"*
⟨*proof*⟩

lemma *sameStoLocTSafe:*
assumes *"TypeSafe ev acc sck mem sto cd"*
shows *"∀struct loc. ((type.Storage struct), (Storeloc loc)) |∈| fmran (Denvalue ev) ⟶ SCon struct loc (sto (Address ev))"* ⟨*proof*⟩

lemma *typeSafeUnique:*
assumes *"TypeSafe ev acc sck mem sto cd"*
shows *"unique_locations (Denvalue ev)"* ⟨*proof*⟩

lemma *typeSafeAccounts:*
assumes *"TypeSafe ev acc sck mem sto cd"*
shows *"balanceTypes acc"* ⟨*proof*⟩

lemma *typeSafeEnvAddresses:*
assumes *"TypeSafe ev acc sck mem sto cd"*
shows *"envAddressesWellFormed ev"* ⟨*proof*⟩

lemma *typeSafeSvalue:*
assumes *"TypeSafe ev acc sck mem sto cd"*
shows *"svalueTypes (Svalue ev)"* ⟨*proof*⟩

lemma *typeSafeCompPointers:*
assumes *"TypeSafe ev acc sck mem sto cd"*
shows *"compPointers sck (Denvalue ev)"* ⟨*proof*⟩

lemma *lexp_stackloc_imps_id:*
assumes *"lexp lv env cd st g = Normal ((LStackloc l2, t'), g')"*
shows *"∃i. lv = Id i"*
⟨*proof*⟩

lemma *lexpDen:*
assumes *"lexp lv env cd (st) g = Normal ((LStackloc l2, t'), g')"*
and *"lv = Id i"*
shows *"(t', Stackloc l2) |∈| fmran (Denvalue env)"*
⟨*proof*⟩

lemma *lexpStackloc_imps_inDen:*
assumes *"lexp lv env cd (st) g = Normal ((LStackloc l2, t'), g')"*
shows *"(t', Stackloc l2) |∈| fmran (Denvalue env)"* ⟨*proof*⟩

lemma *stackLocs_imp_NotDen:*
assumes *"lessThanTopLocs sck'"*
and *"TypeSafe ev' (Accounts st) sck' mem' (Storage st) cd'"*
shows *"∀x y. (Denvalue ev') $$ x = Some y ⟶ snd y ≠ (Stackloc (ShowL$_{nat}$ (Toploc sck')))"*
⟨*proof*⟩

lemma *typeSafe_noDenElementOverToploc_mem:*
assumes *" TypeSafe env (Accounts st) (Stack st) mem (Storage st) cd"*
and *"((type.Memory struct), (Stackloc loc)) |∈| fmran (Denvalue env)"*
and *"(accessStore loc (Stack st)) = Some(KMemptr stloc)"*
shows *"¬LSubPrefL2 stloc (ShowL$_{nat}$ (Toploc mem))"*

⟨*proof*⟩

lemma *typeSafeAllPtrsNotTop:*
assumes *"TypeSafe ev acc sck mem sto cd"*
assumes *"(type.Memory t, Stackloc l) |∈| fmran (Denvalue ev)"*
and *"accessStore l sck = Some (KMemptr ptr)"*
shows *"∀ tl. tl ≥ Toploc mem ⟶*
(¬LSubPrefL2 ptr (ShowL$_{nat}$ tl))∧
(¬LSubPrefL2 (ShowL$_{nat}$ tl) ptr)"
⟨*proof*⟩

lemma *typeSafeAllMemPtrsCantTop:*
assumes *"TypeSafe ev acc sck mem sto cd"*
assumes *"(type.Memory tp, Stackloc l) |∈| fmran (Denvalue ev)"*
and *"accessStore l sck = Some (KMemptr ptr)"*
shows *"∀ len arr loc tl. tl ≥ Toploc mem ∧ tp = MTArray len arr ∧ TypedMemSubPrefPtrs mem len arr ptr loc ⟶*
(¬LSubPrefL2 loc (ShowL$_{nat}$ tl))∧
(¬LSubPrefL2 (ShowL$_{nat}$ tl) loc)"
⟨*proof*⟩

lemma *AccessedMemPtrsCantTop:*
assumes *"lessThanTopLocs mem"*
shows *"∀ len arr loc tl. tl ≥ Toploc mem ∧ tp = MTArray len arr ∧*
MCon (MTArray len arr) mem ptr ∧ TypedMemSubPrefPtrs mem len arr ptr loc ⟶
(¬LSubPrefL2 loc (ShowL$_{nat}$ tl))∧
(¬LSubPrefL2 (ShowL$_{nat}$ tl) loc)"
⟨*proof*⟩

lemma *CompMemTypeSubIndexes:*
assumes *"CompMemType mem len arr (MTArray x t) pParentPtr p"*
and *"subPrefixStructuralConsistency mem"*
and *"MCon (MTArray len arr) mem pParentPtr"*
and *"∀ i<len. accessTypeStore (hash pParentPtr (ShowL$_{nat}$ i)) mem = Some arr"*
shows *"∀ i<x. accessTypeStore (hash p (ShowL$_{nat}$ i)) mem = Some t"* ⟨*proof*⟩

end

A storage pointer derived from a declared storage variable is again related to a declared storage variable.

lemma *CompStoType_ptr_witness:*
assumes *decl: "(type.Storage tp0, Storeloc s0) |∈| fmran den"*
and *base: "CompStoType tp0 t s0 l'"*
and *rel: "t = t' ∧ v = KStoptr l' ∨ extractValueType v ≠ l' ∧ CompStoType t t' l' (extractValueType v)"*
and *ptr: "∃ xx. v = KStoptr xx"*
shows *"∃ stloc tp''. (type.Storage tp'', Storeloc stloc) |∈| fmran den ∧*
(tp'' = t' ∧ v = KStoptr stloc ∨
extractValueType v ≠ stloc ∧ CompStoType tp'' t' stloc (extractValueType v))"
⟨*proof*⟩

end

8 Storage-to-storage copies (TypeSafe_Copies)

theory *TypeSafe_Copies*
imports *TypeSafe_Support TypeSafe_Def*
begin

Locality and consistency properties of *copyRec* and *copy*. A copy changes only sub-locations of its destination and preserves *SCon* at unrelated locations.

context *typesafe_base*
begin

lemma *copyRec_preserves_dest_root:*
shows *"∀t'. (copyRec (hash ls suffixa) (hash ld suffix') t m' = Some v'')⟶ accessStorage t' ld v'' = accessStorage t' ld m'"*
⟨*proof*⟩

lemma *copyrec_SubPrefixes:*
"copyRec srcl destl tp a = Some v''' ⟶
(∀destl' t. (¬LSubPrefL2 destl' destl) ⟶ accessStorage t destl' a = accessStorage t destl' v''')"
⟨*proof*⟩

lemma *copyrec_SubPrefixes_typedPref:*
"copyRec srcl destl tp a = Some v''' ⟶
(∀destl' t. ¬ TypedStoSubpref destl' destl tp ⟶ accessStorage t destl' a = accessStorage t destl' v'''')"
⟨*proof*⟩

lemma *copyrec_SubPrefixes_typedPref_rev:*
"copyRec srcl destl tp a = Some v''' ⟶
(∀destl' t. accessStorage t destl' a ≠ accessStorage t destl' v''' ⟶ TypedStoSubpref destl' destl tp)"
⟨*proof*⟩

lemma *copySingleChange:*
assumes *"iter' (λi m. copyRec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t m) md x = Some updM"*
shows *"∀t' locs. ¬LSubPrefL2 locs ld ∨ locs = ld ⟶ accessStorage t' locs md = accessStorage t' locs updM"*
⟨*proof*⟩

lemma *copySingleChange_typedPref:*
assumes *"iter' (λi m. copyRec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t m) md x = Some updM"*
shows *"∀locs t''. ¬TypedStoSubpref locs ld (STArray x t) ⟶ accessStorage t'' locs md = accessStorage t'' locs updM"*
⟨*proof*⟩

lemma *copyRec_nonZero_Arrays:*
assumes *"copyRec srcl destl t' srcMem = Some v''''"*
and *"subStoTp (STArray len t) t'"*
and *"len = 0"*
shows *"srcMem = v''''"* ⟨*proof*⟩

lemma *iter'Copy_nonZero_Arrays:*
assumes *"iter' (λi. copyRec (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t') srcMem x = Some v'"*
and *"subStoTp (STArray len t) t'"*
and *"len = 0"*
shows *"srcMem = v'"* ⟨*proof*⟩

lemma *copyRec_subT_mustNotBeArray:*
assumes *"copyRec srcl destl t' srcMem = Some v''''"*
and *"∀len t. subStoTp (STArray len t) t' ⟶ len > 0"*
shows *"∀k v. ¬subStoTp (STMap k v) t'"* ⟨*proof*⟩

lemma *copy_unchanged:*
assumes *"iter' (λi. copyRec (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t') srcMem x = Some v'"*
and *"SCon (STArray x t') srcl srcMem"*
and *"SCon (STArray x t') destl srcMem"*
shows *"∀destl' t i' . i'≥x ∧ LSubPrefL2 destl' (hash destl (ShowL$_{nat}$ i')) ⟶*
accessStorage t destl' srcMem = accessStorage t destl' v'" ⟨*proof*⟩

lemma *copy_unchanged_limit:*
assumes *"iter' (λi. copyRec (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t') srcMem x = Some v'"*

shows *"∀destl' . ¬TypedStoSubpref destl' destl (STArray (Suc x) t') ⟶*
(∀t. accessStorage t destl' srcMem = accessStorage t destl' v')"
⟨*proof*⟩

lemma *copy_unchanged_limit_len:*
assumes *"iter' (λi. copyRec (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t') srcMem x = Some v'"*
shows *"∀destl' . ¬TypedStoSubpref destl' destl (STArray (x) t') ⟶*
(∀t. accessStorage t destl' srcMem = accessStorage t destl' v')"
⟨*proof*⟩

lemma *copyRec_unchanged_limit:*
assumes *"copyRec srcl destl t' srcMem = Some v'"*
shows *"∀destl'. ¬TypedStoSubpref destl' destl t' ⟶*
(∀t. accessStorage t destl' srcMem = accessStorage t destl' v')" ⟨*proof*⟩

lemma *copy_unchanged_limit_last_index:*
assumes *"iter' (λi. copyRec (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t') srcMem x = Some v'"*
shows *"∀destl' . TypedStoSubpref destl' (hash destl (ShowL$_{nat}$ x)) t' ⟶*
(∀t. accessStorage t destl' srcMem = accessStorage t destl' v')"
⟨*proof*⟩

lemma *copyRec_notSub_SCon:*
assumes *"copyRec srcl destl t' srcMem = Some v'"*
and *"SCon t' srcl srcMem"*
and *"¬TypedStoSubpref destl srcl t'"*
and *"¬TypedStoSubpref srcl destl t'"*
and *"SCon t' destl srcMem"*
and *"∀subL. TypedStoSubpref subL srcl t' ⟶ ¬ TypedStoSubpref subL destl t'"*
and *"∀subL. TypedStoSubpref subL destl t' ⟶ ¬ TypedStoSubpref subL srcl t'"*
shows *"SCon t' destl v'"* ⟨*proof*⟩

lemma *copy_notSub_SCon:*
assumes *"iter' (λi. copyRec (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t') srcMem x = Some v'"*
and *"SCon (STArray x t') srcl srcMem"*
and *"¬TypedStoSubpref destl srcl (STArray x t')"*
and *"¬TypedStoSubpref srcl destl (STArray x t')"*
and *"SCon (STArray x t') destl srcMem"*
and *"∀subL. TypedStoSubpref subL srcl (STArray x t') ⟶ ¬ TypedStoSubpref subL destl (STArray x t')"*
and *"∀subL. TypedStoSubpref subL destl (STArray x t') ⟶ ¬ TypedStoSubpref subL srcl (STArray x t')"*
shows *"SCon (STArray x t') destl v'"* ⟨*proof*⟩

lemma *copyrec_same_SCon:*
assumes *"SCon t' srcl srcMem"*
and *"copyRec srcl srcl t' srcMem = Some v'''"*
shows *"SCon t' srcl v'''"* ⟨*proof*⟩

lemma *copy_same_SCon:*
assumes *"iter' (λi. copyRec (hash srcl (ShowL$_{nat}$ i)) (hash srcl (ShowL$_{nat}$ i)) t') srcMem x = Some v'"*
and *"SCon (STArray x t') srcl srcMem"*
shows *"SCon (STArray x t') srcl v'"* ⟨*proof*⟩

end
end

9 Memory initialisation (TypeSafe_Memory_Init)

theory *TypeSafe_Memory_Init*
imports *TypeSafe_Support TypeSafe_Storage TypeSafe_Memory*

begin

Properties of *minitRec* and *minit*: locality of the writes, self-pointing array pointers, *MCon* of the initialised region, and preservation of *Toploc*.

9.1 Locality of initialisation

lemma *minitRec_SubPrefixes:*
"minitRec destl tp v' = v''' ⟶
(∀ destl'. (¬LSubPrefL2 destl' destl) ⟶ accessStore destl' v' = accessStore destl' v''')"
⟨*proof*⟩

lemma *minitRec_SubPrefixes_typed:*
"minitRec destl tp v' = v''' ⟶
(∀ destl'. (¬LSubPrefL2 destl' destl) ⟶ accessTypeStore destl' v' = accessTypeStore destl' v''')"
⟨*proof*⟩

lemma *minitSingleChange:*
assumes *"iter (λi m'' . minitRec (hash ld (ShowL$_{nat}$ i)) t m'') m x = updM"*
shows *"∀ t' locs. ¬LSubPrefL2 locs ld ∨ locs = ld ⟶ accessStore locs m = accessStore locs updM"*
⟨*proof*⟩

lemma *minitSingleChange_typed:*
assumes *"iter (λi m'' . minitRec (hash ld (ShowL$_{nat}$ i)) t m'') m x = updM"*
shows *"∀ t' locs. ¬LSubPrefL2 locs ld ∨ locs = ld ⟶ accessTypeStore locs m = accessTypeStore locs updM"*
⟨*proof*⟩

lemma *minitRec_SubPrefixes_typedPref:*
"minitRec destl tp a = v''' ⟶
(∀ destl'. destl' ≠ destl ∧ (¬ TypedMemSubPref destl' destl tp) ⟶ accessStore destl' a = accessStore destl' v''')"
⟨*proof*⟩

lemma *minitRec_SubPrefixes_typedPref_typed:*
"minitRec destl tp a = v''' ⟶
(∀ destl'. destl' ≠ destl ∧ (¬ TypedMemSubPref destl' destl tp) ⟶ accessTypeStore destl' a = accessTypeStore destl' v''')"
⟨*proof*⟩

lemma *minitSingleChange_typedPref:*
assumes *"iter (λi m'' . minitRec (hash ld (ShowL$_{nat}$ i)) t m'') m x = updM"*
shows *"∀ locs. ¬TypedMemSubPref locs ld (MTArray x t) ∨ locs = ld ⟶ accessStore locs m = accessStore locs updM"*
⟨*proof*⟩

lemma *minitSingleChange_typedPref_typed:*
assumes *"iter (λi m'' . minitRec (hash ld (ShowL$_{nat}$ i)) t m'') m x = updM"*
shows *"∀ locs. ¬TypedMemSubPref locs ld (MTArray x t) ∨ locs = ld ⟶ accessTypeStore locs m = accessTypeStore locs updM"*
⟨*proof*⟩

9.2 Array roots and self-pointing pointers

lemma *minitRec_array_root_both:*
assumes *step: "minitRec destl (MTArray x t) m0 = v"*
shows *"accessTypeStore destl v = Some (MTArray x t)*
∧ accessStore destl v = Some (MPointer destl)"
⟨*proof*⟩

lemma *minitRecPtrsPointToSelf:*
shows *"minitRec destl t destm = v'*

```
        ⟶ (∀ l l'. (TypedMemSubPref l destl t ∨ l = destl) ∧ accessStore l v' = Some (MPointer l')
⟶ l' = l)"
```
⟨*proof*⟩

lemma *minitSelfPointers:*
assumes *"iter (λi m''. minitRec (hash destl (ShowL$_{nat}$ i)) t m'') m x1 = v'"*
shows *"(∀ l l'. TypedMemSubPref l destl (MTArray x1 t) ∧ accessStore l v' = Some (MPointer l') ⟶ l' = l)"* ⟨*proof*⟩

9.3 Consistency of initialised memory

lemma *minitRecMCon :*
"minitRec destl tp a = v''' ⟶ arraysGreaterZero tp
⟶ MCon tp v''' destl"
⟨*proof*⟩

lemma *MConMinit:*
assumes *"iter (λi m''. minitRec (hash ld (ShowL$_{nat}$ i)) t m'') md x = updM"*
and *"accessStore (ld) md = None"*
and *"arraysGreaterZero (MTArray x t)"*
shows *" MCon (MTArray x t) updM ld"* ⟨*proof*⟩

9.4 Existing locations and the top location

lemma *minitRecSomeOldNew:*
shows *"minitRec destl t destm = v'*
⟶ (∀ l. (TypedMemSubPref l destl t ∨ l = destl) ⟶ accessStore l destm = None)
⟶ (∀ l accx. accessStore l destm = Some accx ⟶ accessStore l v' = Some accx)"
⟨*proof*⟩

lemma *minitSomeOldNew:*
assumes *"iter (λi m''. minitRec (hash destl (ShowL$_{nat}$ i)) t m'') m x1 = v'"*
and *"∀ l. (TypedMemSubPref l destl (MTArray x1 t) ∨ l = destl) ⟶ accessStore l m = None"*
shows *"(∀ l accx. accessStore l m = Some accx ⟶ accessStore l v' = Some accx)"* ⟨*proof*⟩

lemma *minitRectoploc:*
shows *"minitRec destl t destm = v'*
⟶ Toploc destm = Toploc v'"
⟨*proof*⟩

lemma *minitToploc:*
assumes *"iter (λi m''. minitRec (hash destl (ShowL$_{nat}$ i)) t m'') m x1 = v'"*
shows *"Toploc m = Toploc v'"*
⟨*proof*⟩

9.5 Types of initialised locations

lemma *minit_TypeCompChangeIndexs:*
assumes *"iter (λi m''. minitRec (hash destl (ShowL$_{nat}$ i)) t m'') m x1 = v'"*
shows *"∀ i<x1. accessTypeStore (hash destl (ShowL$_{nat}$ i)) v' = Some t"*
⟨*proof*⟩

lemma *minitRec_TypeCompChange:*
"minitRec destl tp a = v''' ⟶
(∀ destl'.
TypedMemSubPref destl' destl tp ⟶
(case tp of MTValue val ⇒ accessTypeStore destl v''' = Some tp
| MTArray x' t' ⇒ (∃ t''. CompMemType v''' x' t' t'' destl destl' ∧
(case t'' of MTArray parent_len parent_arr ⇒
∀ i<parent_len. accessTypeStore (hash destl' (ShowL$_{nat}$ i)) v''' = Some parent_arr
| MTValue pval ⇒ accessTypeStore destl' v''' = Some (MTValue pval))

```
                        )))"
```
⟨*proof*⟩

lemma *minit_TypeCompChange:*
```
  assumes "iter (λi m''. minitRec (hash destl (ShowL_nat i)) t m'') m x1 = v'"
  shows "(∀ destl'. TypedMemSubPref destl' destl (MTArray x1 t) ⟶
          (∃ st. CompMemType v' x1 t st destl destl' ∧
            (case st of MTArray parent_len parent_arr ⇒
              ∀ i<parent_len. accessTypeStore (hash destl' (ShowL_nat i)) v' = Some parent_arr
             | MTValue pval ⇒ accessTypeStore destl' v' = Some (MTValue pval))))"
```
⟨*proof*⟩

lemma *minitRec_somesome:*
```
  "minitRec destl tp a = v''' ⟶
  (∀ destl'. TypedMemSubPref destl' destl tp ⟶ ((∃ t. accessStore destl' v''' = Some t)
      ⟷ (∃ tt. accessTypeStore destl' v''' = Some tt)))"
```
⟨*proof*⟩

lemma *minit_TypeCompChange_somesome:*
```
  assumes "iter (λi m''. minitRec (hash destl (ShowL_nat i)) t m'') m x1 = v'"
  shows "(∀ destl'. TypedMemSubPref destl' destl (MTArray x1 t) ⟶
          ((∃ t. accessStore destl' v' = Some t) ⟷ (∃ tt. accessTypeStore destl' v' = Some tt)))"
```
⟨*proof*⟩

end

10 Compatibility of storage and memory types (TypeSafe_Mem_Sto_Comp)

theory *TypeSafe_Mem_Sto_Comp*
imports *TypeSafe_Memory TypeSafe_Storage*
begin

Relates *cps2mTypeCompatible* to *cps2mTypeConvert* and transfers typed sub-prefixes between storage and memory types.

lemma *convertComp:*
```
  assumes "cps2mTypeCompatible t' tp"
  shows "cps2mTypeConvert t' = Some tp"
```
⟨*proof*⟩

lemma *compatible_TypedStoSubpref_imps_TypedMemSubPref:*
```
  assumes "cps2mTypeCompatible st mt"
  shows "TypedStoSubpref x' l st ⟷ TypedMemSubPref x' l mt ∨ x' = l"
```
⟨*proof*⟩

lemma *compatible_TypedStoSubpref_imps_TypedMemSubPref_neg:*
```
  assumes "cps2mTypeCompatible st mt"
  shows "¬ TypedStoSubpref x' l st ⟷ ¬ TypedMemSubPref x' l mt ∧ x' ≠ l"
```
⟨*proof*⟩

end

11 Copies between memory and storage (TypeSafe_Memory_Copies)

theory *TypeSafe_Memory_Copies*
imports *TypeSafe_Support TypeSafe_Mem_Sto_Comp*
begin

11.1 Consistency Preservation Under Disjoint Locations

If two locations *destl* and *destl'* are disjoint – neither is a sub-prefix of the other under *LSubPrefL2* – then an update to *destl* cannot affect *MCon* at *destl'*. This lemma is the foundation for all sub-prefix preservation arguments: it shows that the *MCon* invariant is inherited at unrelated locations.

lemma *MCon_preserved_under_disjoint_location:*
assumes *"(¬LSubPrefL2 destl' destl) ∧ ¬LSubPrefL2 destl destl' "*
shows *"(∀ l l'. TypedStoSubpref l destl' t ∧ accessStore l v' = Some (MPointer l') ⟶ l' = l)*
∧ cps2mTypeCompatible t t'
∧ (∀ destl'. ¬ LSubPrefL2 destl' destl ⟶ accessStore destl' v' = accessStore destl' v'')
∧ MCon t' v' destl' ⟶ MCon t' v'' destl'" ⟨*proof*⟩

11.2 Destination Root Preservation

The top-level entry at the destination root location *destl* is unchanged after any recursive copy step. Concretely, *accessStore destl* and *accessTypeStore destl* return the same value before and after the call.

cpm2m works by iterating *iter'* over each array index: at each step it calls *cpm2mrec* on one index, and *cpm2mrec* may recursively call *iter'* for nested arrays. To support the mutual induction the following root-preservation lemmas are established first, covering all three copy directions: *cpm2mrec* (memory to memory), *cps2mrec* (storage to memory), and *cpm2srec* (memory to storage).

updateTypedStore only writes to the single location *destl*, so any *destl'* with *destl' ≠ destl* is unchanged in both value and type stores.

lemma *updateTypedStore_neq_loc:*
"destl' ≠ destl
⟹ accessStore destl' (updateTypedStore destl v t m) = accessStore destl' m
∧ accessTypeStore destl' (updateTypedStore destl v t m) = accessTypeStore destl' m"
⟨*proof*⟩

lemma *cpm2mrec_preserves_dest_root_both:*
shows *"(cpm2mrec (hash ls suffixa) (hash ld suffix') t ms m = Some v'')*
⟶ accessStore ld v'' = accessStore ld m
∧ accessTypeStore ld v'' = accessTypeStore ld m"
⟨*proof*⟩

lemma *cpm2mrec_preserves_dest_root:*
shows *"(cpm2mrec (hash ls suffixa) (hash ld suffix') t ms m = Some v'')⟶ accessStore ld v'' = accessStore ld m"*
⟨*proof*⟩

lemma *cpm2mrec_preserves_dest_root_typed:*
shows *"(cpm2mrec (hash ls suffixa) (hash ld suffix') t ms m = Some v'')⟶ accessTypeStore ld v'' = accessTypeStore ld m"*
⟨*proof*⟩

lemma *cps2mrec_preserves_dest_root_both:*
shows *"(cps2mrec (hash ls suffixa) (hash ld suffix') t ms m' = Some v'')*
⟶ accessStore ld v'' = accessStore ld m'
∧ accessTypeStore ld v'' = accessTypeStore ld m'"
⟨*proof*⟩

lemma *cps2mrec_preserves_dest_root:*
shows *"(cps2mrec (hash ls suffixa) (hash ld suffix') t ms m' = Some v'')⟶ accessStore ld v'' = accessStore ld m'"*
⟨*proof*⟩

lemma *cpm2srec_preserves_dest_root:*
shows *"∀ t'. (cpm2srec (hash ls suffixa) (hash ld suffix') t ms m' = Some v'')⟶ accessStorage t' ld v'' = accessStorage t' ld m'"*
⟨*proof*⟩

11.3 Single-Step Change for Recursive Copy Functions

Each call to *cpm2mrec* or *cps2mrec* (a single recursive copy step) only modifies locations that are sub-prefixes of the destination index being written. Any location whose suffix differs from the destination suffix is left unchanged.

These lemmas underpin the iterated single-change results: by showing the invariant holds at every recursive step, it can be propagated across the full *iter'* loop. The *iterIndexMinOne* helper establishes that a successful iteration of length *x* implies the existence of an intermediate result after *x* steps.

lemma *iterIndexMinOne:*
assumes *"iter' (λi. cpm2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms) md (Suc x) = Some v''"*
shows *"∃v'. iter' (λi. cpm2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms) md x = Some v'"*
⟨*proof*⟩

lemma *cpm2m_iter_dest_unchanged:*
shows *" Some v'' = iter' (λi. cpm2mrec (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t' srcMem) m x1*
∧ (∀suffix. hash destl suffix ≠ destl) ⟶ accessStore destl m = accessStore destl v'' "
⟨*proof*⟩

lemma *cps2mAccessPrePost:*
shows *" Some v'' = iter' (λi. cps2mrec (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t' srcMem) m x1*
∧ (∀suffix. hash destl suffix ≠ destl) ⟶ accessStore destl m = accessStore destl v'' "
⟨*proof*⟩

11.4 Sub-Prefix Preservation: Location-Prefix Condition

A copy operation only modifies locations that are sub-prefixes of the destination root. Any location *destl'* that is not a sub-prefix of *destl* (under *LSubPrefL2*) is left unchanged in both the value and type stores.

This section covers the *both*-access (value and type together), value-only, and typed-only variants for all three recursive copy functions: *cpm2mrec*, *cps2mrec*, and *cpm2srec*. The *cpm2srec_SubPrefixes_typedPref* lemma additionally handles the case where the type compatibility predicate (*cps2mTypeCompatible*) is involved.

The following meta-lemma lifts a pointwise step-preservation property to a full *iter'* run. It abstracts the repeated pattern in every *SubPrefixes_both* proof: the *Not_Sub_More_Specific* chain and invariant bookkeeping are done once here.

lemma *iter'_SubPref_both:*
fixes *destl :: String.literal*
assumes *run: "iter' f init_m x = Some v'"*
assumes *init: "∀loc. ¬LSubPrefL2 loc destl*
⟶ accessStore loc a = accessStore loc init_m
∧ accessTypeStore loc a = accessTypeStore loc init_m"
assumes *step_pres: "⋀i m m'. f i m = Some m'*
⟹ ∀loc. ¬LSubPrefL2 loc (hash destl (ShowL$_{nat}$ i))
⟶ accessStore loc m = accessStore loc m'
∧ accessTypeStore loc m = accessTypeStore loc m'"
shows *"∀loc. ¬LSubPrefL2 loc destl*
⟶ accessStore loc a = accessStore loc v'
∧ accessTypeStore loc a = accessTypeStore loc v'"
⟨*proof*⟩

lemma *cpm2mrec_SubPrefixes_both:*
"cpm2mrec srcl destl tp srcMem a = Some v''' ⟶
(∀destl'. (¬LSubPrefL2 destl' destl) ⟶
accessStore destl' a = accessStore destl' v'''
∧ accessTypeStore destl' a = accessTypeStore destl' v''')"
⟨*proof*⟩

lemma *cpm2mrec_SubPrefixes:*
"cpm2mrec srcl destl tp srcMem a = Some v''' ⟶
(∀destl'. (¬LSubPrefL2 destl' destl) ⟶ accessStore destl' a = accessStore destl' v''')"
⟨*proof*⟩

lemma *cps2mrec_SubPrefixes_both:*
```
  "cps2mrec srcl destl tp srcMem a = Some v''' ⟶
  (∀ destl'. (¬LSubPrefL2 destl' destl) ⟶
      accessStore destl' a = accessStore destl' v'''
    ∧ accessTypeStore destl' a = accessTypeStore destl' v''')"
```
⟨*proof*⟩

lemma *cps2mrec_SubPrefixes:*
```
  "cps2mrec srcl destl tp srcMem a = Some v''' ⟶
  (∀ destl'. (¬LSubPrefL2 destl' destl) ⟶ accessStore destl' a = accessStore destl' v''')"
```
⟨*proof*⟩

lemma *cpm2srec_SubPrefixes:*
```
  "cpm2srec srcl destl tp srcMem a = Some v''' ⟶
  (∀ destl' t. (¬LSubPrefL2 destl' destl) ⟶ accessStorage t destl' a = accessStorage t destl' v''')"
```
⟨*proof*⟩

11.5 Value Type Preservation Under Iterated Copying

When the copied type is a value type (`MTValue`/`STValue`), every destination index written by the iteration holds a value entry (not a pointer) in the resulting memory. These lemmas are used in the `MCon`/`SCon` correctness proofs to discharge the value-branch obligations.

lemma *Cpm2mrec_val_types:*
```
  assumes "iter' (λi. cpm2mrec (hash srcl (ShowL_nat i)) (hash destl (ShowL_nat i)) (MTValue x2) srcMem)
m x1 = Some v'"
  shows "∀ i<x1. (∃ val. accessStore (hash destl (ShowL_nat i)) v' = Some (MValue val))"
```
⟨*proof*⟩

lemma *Cps2mrec_val_types:*
```
  assumes "iter' (λi. cps2mrec (hash srcl (ShowL_nat i)) (hash destl (ShowL_nat i)) (STValue x2) srcMem)
m x1 = Some v'"
  shows "∀ i<x1. (∃ val. accessStore (hash destl (ShowL_nat i)) v' = Some (MValue val))"
```
⟨*proof*⟩

11.6 Pointer Self-Reference Preservation

After a copy, every pointer stored within the destination region points to itself (i.e. the pointer value equals the location at which it is stored). This self-referential invariant reflects the structural layout of arrays: sub-location pointers introduced by `cpm2mrec`/`cps2mrec` always satisfy `accessStore l v' = Some (MPointer l)`, matching the requirement imposed by `MCon`/`SCon`.

Results are given for the recursive step (`cpm2mrecPtrsPointToSelf`, `cps2mrecPtrsPointToSelf`) and lifted to the iterated wrappers (`cpm2mSelfPointers`, `cps2mSelfPointers`).

lemma *cpm2mrecPtrsPointToSelf:*
```
  shows "cpm2mrec srcl destl t srcMem destm = Some v'
         ⟶ (∀ l l'. (TypedMemSubPref l destl t ∨ l = destl) ∧ accessStore l v' = Some (MPointer l')
⟶ l' = l)"
```
⟨*proof*⟩

lemma *cps2mrecPtrsPointToSelf:*
```
  shows "cps2mrec srcl destl t srcMem destm = Some v'
         ⟶ (∀ l l'. (TypedStoSubpref l destl t ∨ l = destl) ∧ accessStore l v' = Some (MPointer l')
⟶ l' = l)"
```
⟨*proof*⟩

lemma *cpm2mSelfPointers:*
```
  assumes "iter' (λi. cpm2mrec (hash srcl (ShowL_nat i)) (hash destl (ShowL_nat i)) t srcMem) m x1 = Some
v'"
  shows "(∀ l l'. TypedMemSubPref l destl (MTArray x1 t) ∧ accessStore l v' = Some (MPointer l') ⟶ l'
= l)" ⟨proof⟩
```

lemma *cps2mSelfPointers:*

assumes *"iter' (λi. cps2mrec (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t srcMem) m x1 = Some v'"*
shows *"(∀ l l'. l ≠ destl ∧ TypedStoSubpref l destl (STArray x1 t) ∧ accessStore l v' = Some (MPointer l') ⟶ l' = l)"* ⟨*proof*⟩

lemma *cpm2mrec_preserves_prior_child_case:*
assumes *iter: "iter' (λi. cpm2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms) m x = Some v'"*
and *step: "cpm2mrec (hash ls (ShowL$_{nat}$ x)) (hash ld (ShowL$_{nat}$ x)) t ms v' = Some v''"*
and *mcon: "MCon (MTArray x t) v' ld"*
and *iless: "i < x"*
shows *"case accessStore (hash ld (ShowL$_{nat}$ i)) v'' of None ⇒ False*
| Some (MValue val) ⇒ (case t of MTValue typ ⇒ MCon t v'' (hash ld (ShowL$_{nat}$ i)) | _ ⇒ False)
| Some (MPointer loc2) ⇒ (case t of MTArray len' arr' ⇒ MCon t v'' loc2 | MTValue Types ⇒ False)"
⟨*proof*⟩

lemma *cps2mrec_preserves_prior_child_case:*
assumes *iter: "iter' (λi. cps2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t' ms) m x = Some v'"*
and *step: "cps2mrec (hash ls (ShowL$_{nat}$ x)) (hash ld (ShowL$_{nat}$ x)) t' ms v' = Some v''"*
and *mcon: "MCon (MTArray x t) v' ld"*
and *compat: "cps2mTypeCompatible t' t"*
and *iless: "i < x"*
shows *"case accessStore (hash ld (ShowL$_{nat}$ i)) v'' of None ⇒ False*
| Some (MValue val) ⇒ (case t of MTValue typ ⇒ MCon t v'' (hash ld (ShowL$_{nat}$ i)) | _ ⇒ False)
| Some (MPointer loc2) ⇒ (case t of MTArray len' arr' ⇒ MCon t v'' loc2 | MTValue Types ⇒ False)"
⟨*proof*⟩

lemma *cpm2srec_preserves_prior_child_scon:*
assumes *step: "cpm2srec (hash ls (ShowL$_{nat}$ x)) (hash ld (ShowL$_{nat}$ x)) t ms v' = Some v''"*
and *scon: "SCon tp (hash ld (ShowL$_{nat}$ i)) v'"*
and *iless: "i < x"*
shows *"SCon tp (hash ld (ShowL$_{nat}$ i)) v''"*
⟨*proof*⟩

11.7 Consistency Implication Lemmas for Recursive Copy Functions

These are the key correctness lemmas for the single-step recursive copy functions. They state that, given a well-formed source, a successful copy produces a well-formed destination.

- *cpm2mrec_imps_MCon_dest*: copying from a *MCon*-consistent memory region to *destl* makes *destl MCon*-consistent in the result.
- *cps2mrec_SCon_imps_MCon_dest*: copying from a *SCon*-consistent storage region into memory makes the destination *MCon*-consistent.
- *cpm2srec_imps_SCon_dest*: copying from a *MCon* memory region into storage makes the destination *SCon*-consistent.

  The proofs proceed by induction on the type structure, with the pointer and value cases handled separately.

lemma *cps2mrec_SCon_imps_MCon_dest:*
"SCon tp srcl srcMem ∧ cps2mTypeCompatible (STArray l tp) (MTArray l t') ∧
cps2mrec srcl destl tp srcMem a = Some v'''
⟶ MCon t' v''' destl
"
⟨*proof*⟩

lemma *cpm2srec_imps_SCon_dest :*
"(∀ l. accessStore srcl srcMem = Some (MPointer l) ⟶ MCon t' srcMem l)

```
  ∧(∀ val. accessStore srcl srcMem = Some (MValue val) ⟶ MCon t' srcMem srcl)
 ∧ cps2mTypeCompatible (STArray l tp) (MTArray l t') ∧
   cpm2srec srcl destl t' srcMem a = Some v'''
         ⟶ SCon tp destl v''''"
```
⟨*proof*⟩

11.8 Top-Level Location Preservation

Any copy operation, recursive or iterated, preserves the top-level location tag (*Toploc*) of the memory. This is a structural invariant: copies only update sub-locations within the destination region and never relocate the overall memory object.

Results are given for both the single-step recursive functions (*cpm2mrec*, *cps2mrec*) and the iterated wrappers (*cpm2m*, *cps2m*).

lemma *cpm2mrecTopLocSame:*
shows *" cpm2mrec srcl destl tp' srcMem a = Some v''' ⟶ Toploc a = Toploc v''''"*
⟨*proof*⟩

lemma *cps2mrecTopLocSame:*
shows *" cps2mrec srcl destl tp' srcMem a = Some v''' ⟶ Toploc a = Toploc v''''"*
⟨*proof*⟩

lemma *cpm2mTopLocSame:*
assumes *"iter' (λi m. cpm2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms m) md x = Some updM"*
and *"x>0"*
shows *"Toploc md = Toploc updM"*
⟨*proof*⟩

lemma *cps2mTopLocSame:*
assumes *"iter' (λi m. cps2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms m) md x = Some updM"*
shows *"Toploc md = Toploc updM"*
⟨*proof*⟩

11.9 Iterated Single-Change Lemmas: Location-Prefix Restriction

These lemmas lift the recursive single-change results to the iterated copies *cpm2m*, *cpm2s*. A location that is not a sub-prefix of the destination root (or is the root itself) is unchanged in both the value and type stores after the full iteration.

The proof uses *iter'_invariant*: the property is maintained as an invariant across every step of the iteration, relying on the recursive single-change lemmas from the previous section. Variants cover memory-to-memory (*cpm2m*), memory-to-storage (*cpm2s*), and separate value/type store projections.

lemma *cpm2mSingleChange:*
assumes *"iter' (λi m. cpm2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms m) md x = Some updM"*
shows *"∀ locs. ¬LSubPrefL2 locs ld ∨ locs = ld ⟶ accessStore locs md = accessStore locs updM"*
⟨*proof*⟩

lemma *cpm2mSingleChange_Typed:*
assumes *"iter' (λi m. cpm2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms m) md x = Some updM"*
shows *"∀ locs. ¬LSubPrefL2 locs ld ∨ locs = ld ⟶ accessTypeStore locs md = accessTypeStore locs updM"*
⟨*proof*⟩

lemma *cpm2sSingleChange:*
assumes *"iter' (λi m. cpm2srec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms m) md x = Some updM"*
shows *"∀ t' locs. ¬LSubPrefL2 locs ld ∨ locs = ld ⟶ accessStorage t' locs md = accessStorage t' locs updM"*
⟨*proof*⟩

lemma *cps2mSingleChange_both:*
assumes *"iter' (λi m. cps2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms m) md x = Some updM"*
shows *"∀ locs. ¬LSubPrefL2 locs ld ∨ locs = ld ⟶ accessStore locs md = accessStore locs updM ∧ accessTypeStore locs md = accessTypeStore locs updM"*
⟨*proof*⟩

lemma *cps2mSingleChange:*
assumes *"iter' (λi m. cps2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms m) md x = Some updM"*
shows *"∀ locs. ¬LSubPrefL2 locs ld ∨ locs = ld ⟶ accessStore locs md = accessStore locs updM"*
⟨*proof*⟩

lemma *cps2mSingleChange_Typed:*
assumes *"iter' (λi m. cps2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms m) md x = Some updM"*
shows *"∀ locs. ¬LSubPrefL2 locs ld ∨ locs = ld ⟶ accessTypeStore locs md = accessTypeStore locs updM"*
⟨*proof*⟩

11.10 Sub-Prefix Preservation: Typed Membership Condition

A second family of sub-prefix preservation lemmas, using *TypedMemSubPref* and *TypedStoSubpref* instead of the plain *LSubPrefL2* relation. These predicates combine location prefix membership with the array-length information encoded in the type, giving a tighter characterisation of which locations are modified.

A location that is not the destination root and is not a typed sub-prefix of the destination is unchanged in both the value and type stores. Results are given for *cpm2mrec*, *cps2mrec* (recursive), and their *both*-access counterparts.

lemma *cpm2mrec_dest_case:*
assumes *step: "cpm2mrec srcl destl t ms md = Some v''"*
and *mcon: "MCon t v'' destl"*
shows *"case accessStore destl v'' of None ⇒ False | Some (MValue val) ⇒ (case t of MTValue typ ⇒ MCon t v'' destl | _ ⇒ False) | Some (MPointer loc2) ⇒ (case t of MTArray len' arr' ⇒ MCon t v'' loc2 | MTValue Types ⇒ False)"*
⟨*proof*⟩

lemma *cps2mrec_dest_case:*
assumes *step: "cps2mrec srcl destl tp ms md = Some v''"*
and *mcon: "MCon t v'' destl"*
shows *"case accessStore destl v'' of None ⇒ False | Some (MValue val) ⇒ (case t of MTValue typ ⇒ MCon t v'' destl | _ ⇒ False) | Some (MPointer loc2) ⇒ (case t of MTArray len' arr' ⇒ MCon t v'' loc2 | MTValue Types ⇒ False)"*
⟨*proof*⟩

lemma *cpm2mrec_SubPrefixes_typedPref_both:*
"cpm2mrec srcl destl tp srcMem a = Some v''' ⟶ (∀ destl'. destl' ≠ destl ∧ (¬ TypedMemSubPref destl' destl tp) ⟶ accessStore destl' a = accessStore destl' v''' ∧ accessTypeStore destl' a = accessTypeStore destl' v''')"
⟨*proof*⟩

lemma *cpm2mrec_SubPrefixes_typedPref:*
"cpm2mrec srcl destl tp srcMem a = Some v''' ⟶ (∀ destl'. destl' ≠ destl ∧ (¬ TypedMemSubPref destl' destl tp) ⟶ accessStore destl' a = accessStore destl' v''')"
⟨*proof*⟩

lemma *cpm2mrec_SubPrefixes_typedPref_typed:*
"cpm2mrec srcl destl tp srcMem a = Some v''' ⟶

```
  (∀ destl'. destl' ≠ destl ∧ (¬ TypedMemSubPref destl' destl tp) ⟶ accessTypeStore destl' a =
accessTypeStore destl' v''')"
  ⟨proof⟩

lemma cps2mrec_SubPrefixes_typedPref_both:
  "cps2mrec srcl destl tp srcMem a = Some v''' ⟶
  (∀ destl'. destl' ≠ destl ∧ ¬ TypedStoSubpref destl' destl tp ⟶
      accessStore destl' a = accessStore destl' v'''
    ∧ accessTypeStore destl' a = accessTypeStore destl' v''')"
⟨proof⟩

lemma cpm2srec_SubPrefixes_typedPref:
  "cpm2srec srcl destl tp srcMem a = Some v''' ⟶
  (∀ destl' tp' t. destl' ≠ destl ∧ cps2mTypeCompatible tp' tp ∧ ¬ TypedStoSubpref destl' destl tp'
      ⟶ accessStorage t destl' a = accessStorage t destl' v''')"
⟨proof⟩
```

11.11 Iterated Single-Change Lemmas: Typed Sub-Prefix Restriction

These lemmas strengthen the earlier single-change results by using the typed sub-prefix predicates (*TypedMemSubPref* and *TypedStoSubpref*) rather than the plain location-prefix relation. A location that is not a typed sub-prefix of the destination (taking into account the array length encoded in the type) is guaranteed to be unmodified.

Variants are provided for value store, type store, and combined (*both*) access, and for all three iterated copy directions: memory-to-memory (*cpm2m*), memory-to-storage (*cpm2s*), and storage-to-memory (*cps2m*).

```
lemma cpm2mSingleChange_typedPref:
  assumes "iter' (λi m. cpm2mrec (hash ls (ShowL_nat i)) (hash ld (ShowL_nat i)) t ms m) md x = Some
updM"
  shows "∀ locs. ¬TypedMemSubPref locs ld (MTArray x t) ⟶ accessStore locs md = accessStore locs
updM"
⟨proof⟩

lemma cpm2mSingleChange_typedPref_typed:
  assumes "iter' (λi m. cpm2mrec (hash ls (ShowL_nat i)) (hash ld (ShowL_nat i)) t ms m) md x = Some
updM"
  shows "∀ locs. ¬TypedMemSubPref locs ld (MTArray x t) ∨ locs = ld ⟶ accessTypeStore locs md =
accessTypeStore locs updM"
⟨proof⟩


lemma cpm2sSingleChange_typedPref:
  assumes "iter' (λi m. cpm2srec (hash ls (ShowL_nat i)) (hash ld (ShowL_nat i)) t ms m) md x = Some
updM"
  shows "∀ locs t' t''. cps2mTypeCompatible (STArray x t') (MTArray x t) ∧ locs ≠ ld
        ∧ ¬TypedStoSubpref locs ld (STArray x t') ⟶ accessStorage t'' locs md = accessStorage t''
locs updM"
⟨proof⟩

lemma cps2mSingleChange_typedPref_both:
  assumes "iter' (λi m. cps2mrec (hash ls (ShowL_nat i)) (hash ld (ShowL_nat i)) t ms m) md x = Some
updM"
  shows "∀ locs. locs ≠ ld ∧ ¬TypedStoSubpref locs ld (STArray x t) ⟶
      accessStore locs md = accessStore locs updM
    ∧ accessTypeStore locs md = accessTypeStore locs updM"
⟨proof⟩

lemma cps2mSingleChange_typedPref:
  assumes "iter' (λi m. cps2mrec (hash ls (ShowL_nat i)) (hash ld (ShowL_nat i)) t ms m) md x = Some
updM"
  shows "∀ locs. locs ≠ ld ∧ ¬TypedStoSubpref locs ld (STArray x t) ⟶ accessStore locs md =
accessStore locs updM"
  ⟨proof⟩
```

lemma *cps2mSingleChange_typedPref_typed:*
assumes *"iter' (λi m. cps2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms m) md x = Some updM"*
shows *"∀ locs. locs ≠ ld ∧ ¬TypedStoSubpref locs ld (STArray x t) ⟶ accessTypeStore locs md = accessTypeStore locs updM"*
⟨*proof*⟩

11.12 Memory and Storage Consistency Propagation Under Iterated Copies

These are the main correctness lemmas for the iterated copy operations. They establish that copying a well-formed (*MCon* or *SCon*) source array into a destination produces a well-formed destination array.

The inductive argument follows the *iter'* structure: at each step the destination has been initialised up to index *len*, the next call to *cpm2mrec* (or *cps2mrec*) writes index *len*, and we must show *MCon* up to *len*. The source is assumed to be *MCon* for the full length throughout.

Also included are helper lemmas about locations that are not typed sub-prefixes of the changed region (*cpm2mNotSubPrefsOfOriginal*, *cpm2mLocationRelatedToTopNotInOrig*, *cps2mNotSubPrefsOfChanged*), which support the inductive step.

lemma *cpm2mrec_imps_MCon_dest :*
"(∀ l. accessStore srcl srcMem = Some (MPointer l) ⟶ MCon tp srcMem l)
∧(∀ val. accessStore srcl srcMem = Some (MValue val) ⟶ MCon tp srcMem srcl) ∧
cpm2mrec srcl destl tp srcMem a = Some v'''
⟶ MCon tp v''' destl"
⟨*proof*⟩

lemma *MCon_cpm2m:*
assumes *"iter' (λi m. cpm2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms m) md x = Some updM"*
and *"MCon (MTArray x t) ms ls"*
and *"(∃p. accessStore (ld) md = Some (MPointer p)) ∨ accessStore (ld) md = None"*
shows *" MCon (MTArray x t) updM ld"* ⟨*proof*⟩

lemma *cps2m:*
assumes *"iter' (λi m. cps2mrec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t' ms m) md x = Some updM"*
and *"SCon (STArray x t') ls ms"*
and *"cps2mTypeCompatible (STArray x t') (MTArray x t)"*
and *"(∃p. accessStore (ld) md = Some (MPointer p)) ∨ accessStore (ld) md = None"*
shows *"MCon (MTArray x t) updM ld"* ⟨*proof*⟩

lemma *MCon_cpm2s:*
assumes *"iter' (λi m. cpm2srec (hash ls (ShowL$_{nat}$ i)) (hash ld (ShowL$_{nat}$ i)) t ms m) md x = Some updM"*
and *"MCon (MTArray x t) ms ls"*
and *"cps2mTypeCompatible (STArray x t') (MTArray x t)"*
shows *"SCon (STArray x t') ld updM"* ⟨*proof*⟩

lemma *cpm2mNotSubPrefsOfOriginal:*
assumes *"MCon (MTArray x11 x12) m' stl2"*
and *"∀ l'. l' ≠ stl1 ⟶ ¬ LSubPrefL2 stl1 l'"*
and *"∀ loc. LSubPrefL2 loc stl1 ⟶ accessStore loc mem' = None"*
and *"¬ LSubPrefL2 stl2 stl1"*
and *"MCon (MTArray x11 x12) mem' stl2"*
and *"∀ locs. ¬ LSubPrefL2 locs stl1 ⟶ accessStore locs mem' = accessStore locs m'"*
and *"∀ l' ≠ stl1. ¬(LSubPrefL2 stl1 l')"*
shows *"¬TypedMemSubPrefPtrs m' x11 x12 stl2 stl1"* ⟨*proof*⟩

lemma *cpm2mLocationRelatedToTopNotInOrig:*
assumes *" LSubPrefL2 intimLoc l"*
and *"¬ LSubPrefL2 stl2 l"*
and *"MCon (MTArray x11 x12) m' stl2"*
and *"∀ loc. LSubPrefL2 loc stl1 ⟶ accessStore loc mem' = None"*

```
    and "MCon (MTArray x11 x12) mem' stl2"
    and "∀ locs. ¬ LSubPrefL2 locs stl1 ⟶ accessStore locs mem' = accessStore locs m'"
    and "∀ l' ≠ l. ¬(LSubPrefL2 l l')"
    and "stl1 = l"
  shows "¬TypedMemSubPrefPtrs m' x11 x12 stl2 intimLoc" ⟨proof⟩


lemma cps2mNotSubPrefsOfChanged:
  assumes "MCon (MTArray x t) m' stl1"
    and "¬ LSubPrefL2 stl2 stl1"
    and "∀ la l'. TypedStoSubpref la stl1 (STArray x' t') ∧ accessStore la m' = Some (MPointer l') ⟶
l' = la"
    and "0 < x"
    and "cps2mTypeCompatible (STArray x' t') (MTArray x t)"
  shows "¬TypedMemSubPrefPtrs m' x t stl1 stl2" ⟨proof⟩
```

11.13 Type Compatibility Under Copying

These lemmas establish how type annotations at sub-locations are updated when copying between memory and storage regions with compatible but structurally distinct types. The key predicates are *TypedMemSubPref* and *TypedStoSubpref*, which identify which locations fall within the typed footprint of a copy destination.

Covered operations: *cpm2mrec*, *cpm2m* (memory-to-memory), *cps2mrec*, *cps2m* (storage-to-memory). The *somesome* variants additionally establish that every typed sub-location of the destination holds a value after the copy.

```
lemma cpm2mrec_array_root_both:
  assumes step: "cpm2mrec srcl destl (MTArray x t) ms md = Some v'''"
  shows "accessTypeStore destl v''' = Some (MTArray x t)
    ∧ accessStore destl v''' = Some (MPointer destl)"
⟨proof⟩

lemma cpm2mrec_child_root_type:
  assumes step: "cpm2mrec srcl destl t ms md = Some v'''"
  shows "accessTypeStore destl v''' = Some t"
⟨proof⟩

lemma cpm2m_TypeCompChangeIndexs:
  assumes "iter' (λi m''. cpm2mrec (hash srcl (ShowL_nat i)) (hash destl (ShowL_nat i)) t ms m'') md x1 =
Some v'"
  shows "∀ i<x1. accessTypeStore (hash destl (ShowL_nat i)) v' = Some t"
⟨proof⟩


lemma cpm2mrec_TypeCompChange:
  "cpm2mrec srcl destl tp ms md = Some v''' ⟶
  (∀ destl'. TypedMemSubPref destl' destl tp ⟶
  (case tp of MTValue val ⇒ accessTypeStore destl v''' = Some tp
      | MTArray x' t' ⇒ (∃ t''. CompMemType v''' x' t' t'' destl destl' ∧
                          (case t'' of MTArray parent_len parent_arr ⇒
                              ∀ i<parent_len. accessTypeStore (hash destl' (ShowL_nat i)) v''' = Some
parent_arr
                           | MTValue pval ⇒ accessTypeStore destl' v''' = Some (MTValue pval))
                        )))"
⟨proof⟩

lemma cpm2m_TypeCompChange:
  assumes "iter' (λi m''. cpm2mrec (hash srcl (ShowL_nat i)) (hash destl (ShowL_nat i)) t ms m'') md x1 =
Some v'"
  shows "(∀ destl'. TypedMemSubPref destl' destl (MTArray x1 t) ⟶
          (∃ st. CompMemType v' x1 t st destl destl' ∧
            (case st of MTArray parent_len parent_arr ⇒
              ∀ i<parent_len. accessTypeStore (hash destl' (ShowL_nat i)) v' = Some parent_arr
             | MTValue pval ⇒ accessTypeStore destl' v' = Some (MTValue pval))))"
⟨proof⟩
```

lemma *cps2mrec_array_root_both:*
assumes *step: "cps2mrec srcl destl (STArray x t) ms md = Some v''''"*
and *conv: "cps2mTypeConvert t = Some tt"*
shows *"accessTypeStore destl v''' = Some (MTArray x tt)*
∧ accessStore destl v''' = Some (MPointer destl)"
⟨*proof*⟩

lemma *cps2mrec_child_root_type:*
assumes *step: "cps2mrec srcl destl t ms md = Some v''''"*
and *comp: "cps2mTypeCompatible (STArray l t) (MTArray l tp)"*
shows *"accessTypeStore destl v''' = Some tp"*
⟨*proof*⟩

lemma *cps2m_TypeCompChangeIndexs:*
assumes *"iter' (λi m''. cps2mrec (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t ms m'') md x1 = Some v'"*
and *"cps2mTypeCompatible (STArray x1 t) (MTArray x1 t')"*
shows *"∀ i<x1. accessTypeStore (hash destl (ShowL$_{nat}$ i)) v' = Some t'"*
⟨*proof*⟩

lemma *cps2mrec_TypeCompChange:*
"cps2mrec srcl destl t' ms md = Some v''' ⟶ cps2mTypeCompatible (STArray l t') (MTArray l tp)⟶
(∀ destl'. TypedMemSubPref destl' destl tp ⟶
(case tp of MTValue val ⇒ accessTypeStore destl v''' = Some tp
| MTArray x' t' ⇒ (∃ t''. CompMemType v''' x' t' t'' destl destl' ∧
(case t'' of MTArray parent_len parent_arr ⇒
∀ i<parent_len. accessTypeStore (hash destl' (ShowL$_{nat}$ i)) v''' = Some parent_arr
| MTValue pval ⇒ accessTypeStore destl' v''' = Some (MTValue pval))
)))"
⟨*proof*⟩

lemma *cps2m_TypeCompChange:*
assumes *"iter' (λi m''. cps2mrec (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t' ms m'') md x1 = Some v'"*
and *"cps2mTypeCompatible (STArray x1 t') (MTArray x1 t)"*
shows *"(∀ destl'. TypedMemSubPref destl' destl (MTArray x1 t) ⟶*
(∃ st. CompMemType v' x1 t st destl destl' ∧
(case st of MTArray parent_len parent_arr ⇒
∀ i<parent_len. accessTypeStore (hash destl' (ShowL$_{nat}$ i)) v' = Some parent_arr
| MTValue pval ⇒ accessTypeStore destl' v' = Some (MTValue pval))))" ⟨*proof*⟩

lemma *cps2mrec_somesome:*
"cps2mrec srcl destl tp ms a = Some v''' ⟶ cps2mTypeCompatible tp tp' ⟶
(∀ destl'. TypedMemSubPref destl' destl tp' ⟶ ((∃ t. accessStore destl' v''' = Some t)
⟷ (∃ tt. accessTypeStore destl' v''' = Some tt)))"
⟨*proof*⟩

lemma *cps2m_TypeCompChange_somesome:*
assumes *"iter' (λi m''. cps2mrec (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t' ms m'') md x1 = Some v'"*
and *"cps2mTypeCompatible (STArray x1 t') (MTArray x1 t)"*
shows *"(∀ destl'. TypedMemSubPref destl' destl (MTArray x1 t) ⟶*
((∃ t. accessStore destl' v' = Some t) ⟷ (∃ tt. accessTypeStore destl' v' = Some tt)))"
⟨*proof*⟩

end

12 Environment initialisation and declarations (TypeSafe_Environment)

theory *TypeSafe_Environment*
  imports *TypeSafe_Def TypeSafe_Memory_Init TypeSafe_Memory_Copies*
begin

context *typesafe_base*
begin

12.1 Initialising an environment results in a type-safe environment

In the call and ecall expressions all calculations are done in a clean environment. To do this *ffold_init* is used to create the environment. The following lemmas demonstrate the initialisation process results in a typesafe environment

12.1.1 Safety of empty environments

lemma *emptyTopLocs:*
  shows *"lessThanTopLocs emptyStore"* ⟨*proof*⟩

lemma *typedEmptyTopLocs:*
  shows *"lessThanTopLocs emptyTypedStore"*
  ⟨*proof*⟩

lemma *emptyEnvSafe:*
  assumes *"ev = emptyEnv addr contrct sendr sval"*
    and *"safeContract acc stor"*
    and *"balanceTypes acc"*
    and *"svalueTypes (Svalue ev)"*
    and *"lessThanTopLocs sck"*
    and *"lessThanTopLocs cd"*
    and *"lessThanTopLocs mem"*
    and *"addressFormat addr"*
    and *"addressFormat sendr"*
    and *"AddressTypes acc"*
    and *"subPrefixStructuralConsistency mem"*
    and *"SomeValSomeTyp mem"*
  shows *"TypeSafe ev acc sck mem stor cd"* ⟨*proof*⟩

12.1.2 Variable initialisation preserves unique locations

lemma *envUpdate_other_unchanged:*
  assumes *"e' = e (|Denvalue := fmupd i ((type.Storage tp),(Storeloc i)) (Denvalue e)|)"*
  shows*"(∀x. x ≠ i ⟶ fmlookup (Denvalue e') x = fmlookup (Denvalue e) x)"*
⟨*proof*⟩

There is an assumption here that the variable being loaded does not already exist in the environment. This is gained from the context of *ffold_init* as we are either loading into an empty environment or if the location already exists then an update function is called so the value is overwritten

lemma *uniqueLocationsPreserved:*
  assumes *"unique_locations (Denvalue e)"*
  assumes *"e' = e (| Denvalue := fmupd i ((type.Storage tp),(Storeloc i)) (Denvalue e) |)"*
    and *"(∀x y. Denvalue e $$ x = Some y ⟶ snd y ≠ Storeloc i)"*
  shows *"unique_locations (Denvalue e')"* ⟨*proof*⟩

lemma *compPointersPreserved:*
  assumes *"compPointers st (Denvalue e)"*
  assumes *"e' = e (| Denvalue := fmupd i ((type.Storage tp), (Storeloc i)) (Denvalue e) |)"*
    and *"SCon tp i (stor (Address e'))"*
    and *"TypeSafe e acc st mem stor cd"*
    and *"(∀x y. Denvalue e $$ x = Some y ⟶ snd y ≠ Storeloc i)"*
    and *"∀x y. Denvalue e $$ x = Some y ⟶ snd y = Storeloc x"*

```
    and "∀ x y t1 t2. x ≠ y ∧ Denvalue e $$ x = Some (type.Storage t1, Storeloc x) ∧ Denvalue e $$ y
= Some (type.Storage t2, Storeloc y)
        ⟶ ¬TypedStoSubpref x y t2 ∧ ¬TypedStoSubpref y x t1"
    and "∀x t1. x ≠ i ∧ Denvalue e $$ x = Some (type.Storage t1, Storeloc x) ⟶ ¬TypedStoSubpref x i
tp ∧ ¬TypedStoSubpref i x t1"
  shows "compPointers st (Denvalue e')"
  ⟨proof⟩

lemma compPointersNonStackUpd:
  assumes "compPointers sck' (Denvalue ev')"
    and "Denvalue e = Denvalue (ev'(|Denvalue := Denvalue ev'(ip $$:= (t, (Stackloc (ShowL_nat (Toploc
sck')))))|))"
    and "∀t'. t ≠ type.Storage t'"
    and "k' = push v sck'"
    and "lessThanTopLocs sck'"
    and "TypeSafe ev' (Accounts st) sck' mem' (Storage st) cd'"
    and "Address ev' = Address e"
  shows "compPointers k' (Denvalue e)" ⟨proof⟩


lemma ffoldInitAllLocsStorage:
  shows "∀e' x y. ffold (init ct) (emptyEnv (Address e2) (Contract e) (Sender e3) sval) xs = e'
         ⟶ ( (Denvalue e') $$ x = Some y ⟶ (snd y) = Storeloc x )

             "
⟨proof⟩

lemma DenvalueChange:
  assumes "e' = e(|Denvalue := Denvalue e(i $$:= y)|)"
    and "z ≠ i"
  shows "Denvalue e' $$ z = Denvalue e $$ z" ⟨proof⟩
```

12.1.3 Initialising a single variable preserves type safety

```
lemma initEmptySafe:
  assumes "TypeSafe e acc st mem stor cd"
  assumes "init ct i e = e'"
    and "ep $$ Contract (e::environment) = Some(ct, dud)"
    and "Type (acc (Address (e::environment))) = Some (atype.Contract (Contract e))"
    and "∀x y. (Denvalue e) $$ x = Some y ⟶ (snd y) = Storeloc x"
    and "∀l y. ∃t1. (Denvalue e) $$ l = Some y ⟶ ct $$ l = Some (Var t1) ∧ (fst y) = type.Storage
t1"
  shows "TypeSafe e' acc st mem stor cd"
⟨proof⟩
```

12.1.4 Initialising multiple variables from an empty environment is type-safe

```
lemma ffoldInitTypeSafe:
  assumes " safeContract acc stor"
    and "balanceTypes acc"
    and "svalueTypes sval"
    and "lessThanTopLocs mem"
    and "ep $$ Contract e = Some(ct, dud)"
    and "addressFormat (e2)"
    and "addressFormat (e3)"
    and "AddressTypes acc"
    and "subPrefixStructuralConsistency mem"
    and "SomeValSomeTyp mem"
    and "Type (acc e2) = Some (atype.Contract (Contract e))"
  shows "∀e' x y. ffold (init ct) (emptyEnv (e2) (Contract e) (e3) sval) xs = e'
         ⟶ TypeSafe e' acc emptyStore mem stor emptyTypedStore
             ∧( (Denvalue e') $$ x = Some y ⟶ (snd y) = Storeloc x )
             ∧ ( Denvalue e' $$ x = Some y ⟶ (∃t1. ct $$ x = Some (Var t1) ∧ fst y = type.Storage
t1))
```

"

⟨*proof*⟩

lemma *ffoldInit_var_storage_mapping:*
assumes *"ffold (init ct) (emptyEnv adv c adde v') (fmdom ct) = loaded"*
shows *"∀ id v. ct $$ id = Some (Var v) ⟶ (Denvalue loaded $$ id = Some (type.Storage v, Storeloc id))"* ⟨*proof*⟩

lemma *ffoldInit_var_storage_mapping_rev:*
assumes *"ffold (init ct) (emptyEnv adv c (adde) v') (fmdom ct) = loaded"*
shows *"∀ id v. (Denvalue loaded $$ id = Some (type.Storage v, Storeloc id)) ⟶ ct $$ id = Some (Var v)"* ⟨*proof*⟩

lemma *ffoldInit_var_storage_mapping_eq:*
assumes *"ffold (init ct) (emptyEnv adv c (adde) v') (fmdom ct) = loaded"*
shows *"∀ id v. (Denvalue loaded $$ id = Some (type.Storage v, Storeloc id)) ⟷ ct $$ id = Some (Var v)"*
⟨*proof*⟩

end

end

13 Declarations (TypeSafe_Decl)

theory *TypeSafe_Decl*
imports *TypeSafe_Environment*
begin

context *typesafe_base*
begin

13.1 Declarations are type-safe

decl is used by *local.load* to declare the contract variables in the newly created environment. For simple cases (Stack values) decl will copy the value from the source location into the top of the Stack. For more complex data types such as copying a Memory structure the structure is copied recursively using a support function (such as *cpm2mrec*)

lemma *cdMemLocsToploc:*
assumes *"lessThanTopLocs cd'"*
and *"l = (ShowL$_{nat}$ (Toploc cd'))"*
and *"∃ dud. (dud, c') = allocate cd'"*
and *"Toploc c' = Toploc c"*
and *"∀ locs. ¬ LSubPrefL2 locs l ⟶ accessStore locs cd' = accessStore locs c"*
shows *"lessThanTopLocs c"* ⟨*proof*⟩

13.1.1 Type safety of *decl*

Given a type-safe source environment, a type-consistent value to copy, and a successful *decl*, the result of *decl* is type-safe.

Postcondition of *decl* without an initial value.

abbreviation *declNonePost :: "environment ⇒ stack ⇒ calldataT ⇒ memoryT ⇒ 'a state_scheme ⇒ environment ⇒ calldataT ⇒ memoryT ⇒ stack ⇒ bool"*
where *"declNonePost ev' sck' cd' mem' st e c m' k' ≡ TypeSafe e (Accounts st) k' m' (Storage st) c*

∧ (∀ x l. (∄ y y'. x = type.Memory y ∨ x = Value y') ∧ (x, l) |∈| fmran(Denvalue e) ⟶ (x,l)|∈| fmran(Denvalue ev'))
∧ (∀ sckl ptr. accessStore sckl k' = Some (ptr) ∧ (∄ y y'. ptr = KMemptr y ∨ ptr = KValue y') ⟶ accessStore sckl sck' = Some ptr)
∧ cd' = c
∧ Toploc mem' ≤ Toploc m'

```
∧ (∀ locs v. accessStore locs mem' = Some v ⟶ accessStore locs m' = Some v)
∧ (∀ locs t. accessTypeStore locs mem' = Some t ⟶ accessTypeStore locs m' = Some t)
∧ (∀ locs. (∃ tloc<Toploc mem'. LSubPrefL2 locs (ShowL_nat (tloc)) ∧ accessStore locs mem' = None) ⟶
accessStore locs m' = None)"

context
  fixes c cd cd' cp e ev ev' ip k' m' mem' sck' st tp
  assumes typeSafeDeclNone_hyps: "TypeSafe ev' (Accounts st) sck' mem' (Storage st ) cd'"
      "decl ip tp None cp cd (Memory st) (Storage st (Address ev)) (cd', mem', sck', ev') = Some (c, m',
k', e)"
begin
```

A value-type variable declared without an initial value: its default value is pushed on the stack.

```
lemma typeSafeDeclNone_1:
  assumes "tp = Value t"
    and "None = None"
    and "cp = cp0"
    and "cd = cd0"
    and "state.Memory st = mem0"
    and "state.Storage st (Address ev) = sto0"
    and "(cd', mem', sck', ev') = (c', m, k, e')"
    and "Some (c, m', k', e) = Some (c', m, astack_dup ip (Value t) (KValue (ival t)) (k, e'))"
  shows "declNonePost ev' sck' cd' mem' st e c m' k'"
⟨proof⟩
```

A memory array declared without an initial value: a fresh array is allocated and initialised with `minit`.

```
lemma typeSafeDeclNone_18:
  assumes "tp = type.Memory (MTArray x t)"
    and "None = None"
    and "cp = cp0"
    and "cd = cd0"
    and "state.Memory st = mem0"
    and "state.Storage st (Address ev) = sto0"
    and "(cd', mem', sck', ev') = (c', m, k, e')"
    and "Some (c, m', k', e) = (if Denvalue e' $$ ip = None ∧ arraysGreaterZero (MTArray x t) then let
m' = minit x t m in Some (c',
        m', astack_dup ip (type.Memory (MTArray x t)) (KMemptr (ShowL_nat (Toploc m))) (k, e')) else
None)"
  shows "declNonePost ev' sck' cd' mem' st e c m' k'"
⟨proof⟩


lemma typeSafeDeclNone:
  shows "declNonePost ev' sck' cd' mem' st e c m' k'" ⟨proof⟩



end

definition ncpElementsNoSubPref::"memoryT ⇒ memoryT ⇒ bool"
  where "ncpElementsNoSubPref lm0 lmD = (∀ i l1 t1 l2 t2 loc i2 loc2. Toploc lm0 > i
                                          ∧ Toploc lm0 ≤ i2
                                          ∧ LSubPrefL2 loc (ShowL_nat i)
                                          ∧ LSubPrefL2 loc2 (ShowL_nat i2)
                                          ∧ MCon (MTArray l1 t1) lm0 loc ⟶
                                    ¬TypedMemSubPrefPtrs lmD l2 t2 loc2 loc
                                          ∧ ¬TypedMemSubPrefPtrs lmD l1 t1 loc loc2)"

definition ncpOMemInDMem::"memoryT ⇒ memoryT ⇒ bool"
  where "ncpOMemInDMem lm0 lmD = (∀ i loc. i < Toploc lm0 ∧ LSubPrefL2 loc (ShowL_nat i)
                                  ⟶ accessStore loc lm0 = accessStore loc lmD ∧ accessTypeStore loc
lm0 = accessTypeStore loc lmD)"
```

Every memory reference in `lev0` (the extended environment) that is below the top location of the original memory `mem` already existed in the original environment `lev` and stack `lk`.

definition *ncpDenvalueLimit::"environment ⇒ environment ⇒ stack ⇒ stack ⇒ memoryT ⇒ bool "*
where *"ncpDenvalueLimit lev0 lev lk lst mem = (∀ tp' locs p i. (type.Memory tp', Stackloc locs) |∈| fmran (Denvalue lev0)*
∧ accessStore locs lk = Some (KMemptr p) ∧ i < Toploc mem ∧ LSubPrefL2 p (ShowL$_{nat}$ i)⟶
(∃ tp'' loc2 p'.((type.Memory tp'', Stackloc loc2) |∈| fmran (Denvalue lev)
∧ accessStore loc2 lst = Some (KMemptr p') ∧
((p' = p ∧ tp'' = tp') ∨ (∃ len arr. p' ≠ p ∧ tp'' = MTArray len arr ∧ CompMemType mem len arr tp' p' p))))

)"

definition *ncpNewSelfPoint::"memoryT ⇒ memoryT ⇒ bool"*
where *"ncpNewSelfPoint lm0 lmD = (∀ i loc loc2. Toploc lmD > i*
∧ Toploc lm0 ≤ i
∧ LSubPrefL2 loc (ShowL$_{nat}$ i)
∧ accessStore loc lmD = Some (MPointer loc2)
⟶ loc = loc2
)"

lemma *ncpElementsNoSubPref_sameMem:*
assumes *"TypeSafe e (Accounts st) (Stack st) (Memory st) (Storage st) cd"*
shows *"ncpElementsNoSubPref (Memory st) (Memory st)"* ⟨*proof*⟩

lemma *cpm2mrec_somesome:*
"cpm2mrec srcl destl tp ms a = Some v''' ⟶
(∀ destl'. TypedMemSubPref destl' destl tp ⟶ ((∃ t. accessStore destl' v''' = Some t)
⟷ (∃ tt. accessTypeStore destl' v''' = Some tt)))"
⟨*proof*⟩

lemma *cpm2m_TypeCompChange_somesome:*
assumes *"iter' (λi m''. cpm2mrec (hash srcl (ShowL$_{nat}$ i)) (hash destl (ShowL$_{nat}$ i)) t ms m'') md x1 = Some v'"*
shows *"(∀ destl'. TypedMemSubPref destl' destl (MTArray x1 t) ⟶*
((∃ t. accessStore destl' v' = Some t) ⟷ (∃ tt. accessTypeStore destl' v' = Some tt)))"
⟨*proof*⟩

Postcondition of *decl* with an initial value.

abbreviation *declPost :: "environment ⇒ 'a state_scheme ⇒ bool ⇒ memoryT ⇒ environment ⇒ calldataT ⇒ memoryT ⇒ stack ⇒ bool"*
where *"declPost ev st cp mem' e c m' k' ≡ TypeSafe e (Accounts st) k' m' (Storage st) c ∧*
(¬cp ⟶ (∀ locs tp. MCon tp (Memory st) locs ⟶ MCon tp m' locs)
∧ Toploc (Memory st) ≤ Toploc (m')
∧ ncpDenvalueLimit e ev k' (Stack st) (Memory st)
∧ ncpOMemInDMem (Memory st) m'
∧ ncpElementsNoSubPref (Memory st) m'
∧ ncpNewSelfPoint (Memory st) m'
) ∧ Toploc m' ≥ Toploc mem'"

context
fixes *c cd cd' cp e ev ev' ip k' m' mem' sck' st t'' tp v*
assumes *typeSafeDecl_hyps: "TypeSafe ev (Accounts st) (Stack st) (Memory st) (Storage st) cd"*
"valueCon ev cd st v t''"
"TypeSafe ev' (Accounts st) sck' mem' (Storage st) cd'"
"decl ip tp (Some (v,t'')) cp cd (Memory st) (Storage st (Address ev)) (cd', mem', sck', ev') = Some (c, m', k', e)"
"¬cp ⟶ (∀ locs tp. MCon tp (Memory st) locs ⟶ MCon tp mem' locs) ∧ Toploc (Memory st) ≤ Toploc (mem')
∧ ncpDenvalueLimit ev' ev sck' (Stack st) (Memory st)
∧ ncpOMemInDMem (Memory st) mem'
∧ ncpElementsNoSubPref (Memory st) mem'

```
                ∧ ncpNewSelfPoint (Memory st) mem''"
      "∀struct.
       t'' = type.Memory struct ⟶
       (∃stloc tp'' p. (type.Memory tp'', Stackloc stloc) |∈| fmran (Denvalue ev) ∧ accessStore stloc
(Stack st) = Some (KMemptr p) ∧ (tp'' = struct ∧ v = (KMemptr p)
      ∨ (∃len arr. p ≠ (extractValueType v) ∧ tp'' = MTArray len arr ∧ CompMemType (Memory st) len
arr struct p (extractValueType v))))"
      "∀struct.
       t'' = Calldata struct ⟶
       (∃stloc tp'' p. (Calldata tp'', Stackloc stloc) |∈| fmran (Denvalue ev) ∧ accessStore stloc
(Stack st) = Some (KCDptr p) ∧ (tp'' = struct ∧ v = (KCDptr p)
      ∨ (∃len arr. p ≠ (extractValueType v) ∧ tp'' = MTArray len arr ∧ CompMemType cd len arr struct
p (extractValueType v))))"
      "∀struct. tp = type.Storage struct ⟶ (∀locs tp. SCon tp locs (Storage st (Address ev))
        ⟶ SCon tp locs (Storage st (Address ev'))) ∧ (∃loc tp'' p. (type.Storage tp'', loc) |∈|
fmran
        (Denvalue ev')
                                              ∧(case loc of Stackloc stloc ⇒
                                                  (accessStore stloc sck' =Some (KStoptr
p) ∧ ((tp'' = struct ∧ v = KStoptr p) ∨
                                                  ((extractValueType v) ≠ p ∧
CompStoType tp'' struct p (extractValueType v))))
                                                | Storeloc stloc ⇒
                                                  (((tp'' = struct ∧ v = KStoptr stloc)
∨
                                                  ((extractValueType v) ≠ stloc ∧
CompStoType tp'' struct stloc (extractValueType v))))
                                                ))"
```

begin

A value-type variable initialised with a value, converted to the declared type.

lemma *typeSafeDecl_2:*
assumes "tp = Value t"
and "Some (v, t'') = Some (KValue vc, Value t')"
and "cp = cp0"
and "cd = cd0"
and "state.Memory st = mem0"
and "state.Storage st (Address ev) = sto0"
and "(cd', mem', sck', ev') = (cdc', mem'', sck'', ev'')"
and "Some (c, m', k', e) = convert t' t vc ⤜ (λv'. Some (cdc', mem'', astack_dup ip (Value t)
(KValue v') (sck'', ev'')))"
shows "declPost ev st cp mem' e c m' k'"
⟨*proof*⟩

A calldata array initialised from a calldata pointer with copying: the array is copied into the new calldata.

lemma *typeSafeDecl_9:*
assumes "tp = Calldata (MTArray x t)"
and "Some (v, t'') = Some (KCDptr p, tv0)"
and "cp"
and "cd = cdc"
and "state.Memory st = mem0"
and "state.Storage st (Address ev) = sto0"
and "(cd', mem', sck', ev') = (cdc', mem'', sck'', ev'')"
and "Some (c, m', k', e) = (if tv0 = Calldata (MTArray x t) ∧ 0 < x ∧ Denvalue
ev'' $$ ip = None then let l = (ShowL$_{nat}$ (Toploc cdc')); (uub, c') = allocate cdc' in cpm2m p l
x t cdc c'
⤜ (λc''. Some (c'', mem'', astack_dup ip (Calldata (MTArray x t)) (KCDptr l) (sck'',
ev''))) else None)"
shows "declPost ev st cp mem' e c m' k'"
⟨*proof*⟩

A calldata array initialised from a memory pointer with copying: the array is copied into the new calldata.

lemma *typeSafeDecl_10:*
assumes "tp = Calldata (MTArray x t)"

```
    and "Some (v, t'') = Some (KMemptr p, tv0)"
    and "cp"
    and "cd = cd0"
    and "state.Memory st = mem"
    and "state.Storage st (Address ev) = sto0"
    and "(cd', mem', sck', ev') = (cdc', mem''', sck'', ev'')"
    and "Some (c, m', k', e) = (if tv0 = type.Memory (MTArray x t) ∧ 0 < x ∧ Denvalue
        ev'' $$ ip = None then let l = (ShowL_nat (Toploc cdc')); (uub, c') = allocate cdc' in cpm2m p l
x t mem c'
            ≫= (λc''. Some (c'', mem''', astack_dup ip (Calldata (MTArray x t)) (KCDptr l) (sck'',
ev''))) else None)"
  shows "declPost ev st cp mem' e c m' k'"
⟨proof⟩
```

A memory array initialised from a memory pointer with copying: the array is copied into fresh memory.

```
lemma typeSafeDecl_19:
  assumes "tp = type.Memory (MTArray x t)"
    and "Some (v, t'') = Some (KMemptr p, tv0)"
    and "cp"
    and "cd = cd0"
    and "state.Memory st = mem"
    and "state.Storage st (Address ev) = sto0"
    and "(cd', mem', sck', ev') = (cdc', mem''', sck'', ev'')"
    and "Some (c, m', k', e) = (if tv0 = type.Memory (MTArray x t) ∧ 0 < x ∧ Denvalue
        ev'' $$ ip = None then cpm2m p (ShowL_nat (Toploc mem''')) x t mem (snd (allocate
        mem''')) ≫= (λm'. Some (cdc', m', astack_dup ip (type.Memory (MTArray x t)) (KMemptr (ShowL_nat
(Toploc mem''')))) (sck'', ev''))) else None)"
  shows "declPost ev st cp mem' e c m' k'"
⟨proof⟩
```

A memory array initialised from a memory pointer without copying: the variable refers to the existing array.

```
lemma typeSafeDecl_20:
  assumes "tp = type.Memory (MTArray x t)"
    and "Some (v, t'') = Some (KMemptr p, tv0)"
    and "¬ cp"
    and "cd = cd0"
    and "state.Memory st = mem0"
    and "state.Storage st (Address ev) = sto0"
    and "(cd', mem', sck', ev') = (cdc', mem''', sck'', ev'')"
    and "Some (c, m', k', e) = (if tv0 = type.Memory (MTArray x t) ∧ 0 < x ∧ Denvalue ev'' $$ ip =
None then Some (cdc', mem''',
        astack_dup ip (type.Memory (MTArray x t)) (KMemptr p) (sck'', ev'')) else None)"
  shows "declPost ev st cp mem' e c m' k'"
⟨proof⟩
```

A memory array initialised from a calldata pointer: the array is copied into fresh memory.

```
lemma typeSafeDecl_21:
  assumes "tp = type.Memory (MTArray x t)"
    and "Some (v, t'') = Some (KCDptr p, tv0)"
    and "cp = cp0"
    and "cd = cdc"
    and "state.Memory st = mem0"
    and "state.Storage st (Address ev) = sto0"
    and "(cd', mem', sck', ev') = (cdc', mem''', sck'', ev'')"
    and "Some (c, m', k', e) = (if tv0 = Calldata (MTArray x t) ∧ 0 < x ∧ Denvalue
        ev'' $$ ip = None then cpm2m p (ShowL_nat (Toploc mem''')) x t cdc (snd (allocate
        mem''')) ≫= (λm'. Some (cdc', m', astack_dup ip (type.Memory (MTArray x t)) (KMemptr (ShowL_nat
(Toploc mem''')))) (sck'', ev''))) else None)"
  shows "declPost ev st cp mem' e c m' k'"
⟨proof⟩
```

A memory array initialised from a storage pointer: the storage array is copied into fresh memory with *cps2m*.

```
lemma typeSafeDecl_22:
  assumes "tp = type.Memory (MTArray x t)"
```

```
    and "Some (v, t'') = Some (KStoptr p, type.Storage (STArray x' t'))"
    and "cp = cp0"
    and "cd = cd0"
    and "state.Memory st = mem0"
    and "state.Storage st (Address ev) = s"
    and "(cd', mem', sck', ev') = (cdc', mem''', sck'', ev'')"
    and "Some (c, m', k', e) = (if cps2mTypeCompatible (STArray x' t') (MTArray x t) ∧ Denvalue
        ev'' $$ ip = None then cps2m p (ShowL_nat (Toploc mem''')) x' t' s (snd (allocate
        mem''')) ≫= (λm''. Some (cdc',
        m'', astack_dup ip (type.Memory (MTArray x t)) (KMemptr (ShowL_nat (Toploc mem''')))) (sck'',
ev'''))) else None)"
  shows "declPost ev st cp mem' e c m' k'"
⟨proof⟩
```

A storage array initialised from a storage pointer: the variable refers to the storage array.

```
lemma typeSafeDecl_42:
  assumes "tp = type.Storage (STArray x t)"
    and "Some (v, t'') = Some (KStoptr p, tv0)"
    and "cp = cp0"
    and "cd = cd0"
    and "state.Memory st = mem0"
    and "state.Storage st (Address ev) = sto0"
    and "(cd', mem', sck', ev') = (cdc', mem'', sck'', ev'')"
    and "Some (c, m', k', e) = (if tv0 = type.Storage (STArray x t) ∧ Denvalue ev'' $$ ip = None then
Some (cdc', mem'', astack_dup ip
        (type.Storage (STArray x t)) (KStoptr p) (sck'', ev'')) else None)"
  shows "declPost ev st cp mem' e c m' k'"
⟨proof⟩
```

A storage mapping initialised from a storage pointer: the variable refers to the mapping.

```
lemma typeSafeDecl_43:
  assumes "tp = type.Storage (STMap t t')"
    and "Some (v, t'') = Some (KStoptr p, tv0)"
    and "cp = cp0"
    and "cd = cd0"
    and "state.Memory st = mem0"
    and "state.Storage st (Address ev) = sto0"
    and "(cd', mem', sck', ev') = (cdc', mem'', sck'', ev'')"
    and "Some (c, m', k', e) = (if tv0 = type.Storage (STMap t t') ∧ Denvalue ev'' $$ ip = None then
Some (cdc', mem'', astack_dup ip
        (type.Storage (STMap t t')) (KStoptr p) (sck'', ev'')) else None)"
  shows "declPost ev st cp mem' e c m' k'"
⟨proof⟩


lemma typeSafeDecl:
  shows "declPost ev st cp mem' e c m' k'" ⟨proof⟩


end

end

end
```

14 Expression preservation (TypeSafe_Expressions)

```
theory TypeSafe_Expressions
  imports TypeSafe_Copies TypeSafe_Decl
begin
```

The supported Solidity expressions do not violate the type consistency of a given environment.

```
context typesafe_base
begin
```

14.1 Olift and Plift

lemma *oliftTypeInput:*
assumes *"olift op t1 t2 v1 v2 = Some(v',t')"*
shows *"(∃b. t1 = (TSInt b) ∨ t1 = (TUInt b)) ∧ (∃b2. t2 = (TSInt b2) ∨ t2 = (TUInt b2))"*
⟨*proof*⟩

lemma *pliftTypeInput:*
assumes *"plift op t1 t2 v1 v2 = Some(v',t')"*
shows *"(∃b. t1 = (TSInt b) ∨ t1 = (TUInt b)) ∧ (∃b2. t2 = (TSInt b2) ∨ t2 = (TUInt b2))"*
⟨*proof*⟩

lemma *vtandTypeOut:*
assumes *"vtand t1 t2 v1 v2 = Some(v', t')"*
shows *"(v' = ShowL$_{bool}$ True ∨ v' = ShowL$_{bool}$ False) ∧ t' = TBool"*
⟨*proof*⟩

lemma *vtorTypeOut:*
assumes *"vtor t1 t2 v1 v2 = Some(v', t')"*
shows *"(v' = ShowL$_{bool}$ True ∨ v' = ShowL$_{bool}$ False) ∧ t' = TBool"*
⟨*proof*⟩

lemma *vtandTypeCon:*
assumes *"vtand t1 t2 v1 v2 = Some(v', t')"*
and *"(v' = ShowL$_{bool}$ True ∨ v' = ShowL$_{bool}$ False) ∧ t' = TBool"*
shows *"typeCon t' v'"* ⟨*proof*⟩

14.1.1 Results of successful olift must be typecon

lemma *oliftTypeCon:*
assumes *"olift op t1 t2 v1 v2 = Some(v',t')"*
and *"t1 = TSInt b1 ∨ t1 = TUInt b1"*
and *"t2 = TSInt b2 ∨ t2 = TUInt b2"*
shows *"typeCon t' v'"*
⟨*proof*⟩

14.1.2 Results of successful plift must be typecon

lemma *pliftTypeCon:*
assumes *"plift op t1 t2 v1 v2 = Some(v',t')"*
and *"t1 = TSInt b1 ∨ t1 = TUInt b1"*
and *"t2 = TSInt b2 ∨ t2 = TUInt b2"*
shows *"typeCon t' v'"*
⟨*proof*⟩

14.1.3 Lifting an expression with olift yields typecon

lemma *oliftSuccess:*
assumes *"lift expr (olift op) e1 e2 ev cd st g = Normal ((KValue v, Value t), g')"*
shows *"typeCon t v"*
⟨*proof*⟩

14.1.4 Lifting an expression with plift yields typecon

lemma *pliftSuccess:*
assumes *"lift expr (plift op) e1 e2 ev cd st g = Normal ((KValue v, Value t), g')"*
shows *"typeCon t v"*
⟨*proof*⟩

14.2 Proof that expressions do not violate type consistency

The following lemma proves that given a typesafe environment all expressions, Memory selections, Storage selections and rexpressions return type consistent values, i.e. the returned string values conform to the expected types. This demonstrates that the semantics of expressions do not alter the values in any way which may violate type

consistency. Further, the environment, memory, calldata, and storage created by `local.load` are type-safe with respect to one another.

The environment is assumed to be fully initialised by `ffold_init`: the contract exists in the accounts and all contract variables are initialised. This holds in the context of statements.

lemma *exprTypeconInduct_aux:*

```
  "∀l1' t1' g1' arr. msel c1 t1 l1 xe1 ev1 cd1 st1 g1 = Normal ((l1', t1'), g1') ∧
                          MCon t1 (if c1 then Memory st1 else cd1) l1 ∧
                          TypeSafe ev1 (Accounts st1) (Stack st1) (Memory st1) (Storage st1) cd1 ∧
                          fullyInitialised ev1 (Accounts st1) (Stack st1) ⟶
                          (xe1 ≠ Nil ⟶ (∃len arr. (t1 = MTArray len arr) ∧
                                          (case t1' of MTValue val ⇒ CompMemType (if c1 then Memory
st1 else cd1) len arr t1' l1 l1'
                                                                  |(MTArray l' ar') ⇒ (∃p. accessStore l1' (if
c1 then Memory st1 else cd1) = Some (MPointer p) ∧
                                                                                        CompMemType (if c1 then
Memory st1 else cd1) len arr t1' l1 p))))"
  "∀l2' v2' t2' g2'. ssel t2 l2 xe2 ev2 cd2 st2 g2 = Normal ((l2', t2'), g2') ∧
                          SCon t2 l2 (Storage st2 (Address ev2)) ∧
                          TypeSafe ev2 (Accounts st2) (Stack st2) (Memory st2) (Storage st2) cd2 ∧
                          fullyInitialised ev2 (Accounts st2) (Stack st2) ⟶
                          (xe2 ≠ Nil ⟶ CompStoType t2 t2' l2 l2')"
  "∀v t g4'. expr e4 ev4 cd4 st4 g4 = Normal ((v, t), g4') ∧
                TypeSafe ev4 (Accounts st4) (Stack st4) (Memory st4) (Storage st4 ) cd4 ∧
                  fullyInitialised ev4 (Accounts st4) (Stack st4) ⟶
                    (case t of Value typ ⇒ (typeCon typ (extractValueType(v)) ∧ (∃xx. v = KValue xx))
                     | Calldata struct ⇒ (MCon struct cd4 (extractValueType(v)) ∧ (∃xx. v = KCDptr xx)
                                           ∧ (∃stloc tp'' p. (Calldata tp'', Stackloc stloc) |∈|
fmran (Denvalue ev4)∧ accessStore stloc (Stack st4) =Some (KCDptr p)
                                           ∧ ((tp'' = struct ∧ v = KCDptr p) ∨
                                            (∃len arr. (extractValueType v) ≠ p ∧ tp'' = MTArray len
arr ∧ CompMemType cd4 len arr struct p (extractValueType v)))))
                     | type.Memory struct ⇒ (MCon struct (Memory st4) (extractValueType(v)) ∧ (∃xx. v
= KMemptr xx) ∧
                                          (∃stloc tp'' p. (type.Memory tp'', Stackloc stloc) |∈| fmran
(Denvalue ev4)∧ accessStore stloc (Stack st4) =Some (KMemptr p)
                                           ∧
                                               ((tp'' = struct ∧ v = (KMemptr p)) ∨ (∃len arr.
(extractValueType v)
                                                   ≠ p ∧tp'' = MTArray len arr ∧ CompMemType
(Memory st4) len arr struct
                                                   p (extractValueType v)))))
                     | type.Storage struct ⇒ (SCon struct (extractValueType(v)) (Storage st4 (Address
ev4))) ∧ (∃xx. v = KStoptr xx)
                                            ∧ (∃stloc tp''. (type.Storage tp'', Storeloc stloc) |∈|
fmran (Denvalue ev4)
                                            ∧((((tp'' = struct ∧ v = KStoptr stloc) ∨
                                                        ((extractValueType v) ≠ stloc ∧
CompStoType tp'' struct stloc (extractValueType v))))
                                                      )))"

"∀ev cd k m g'. load lcp lis lxs lev0 lcd0 lk lm lev lcd lst lg = Normal ((ev, cd, k, m), g') ∧
                    (¬lcp ⟶ (∀locs tp. MCon tp (Memory lst) locs ⟶ MCon tp lm locs)
                                    ∧ Toploc (Memory lst) ≤ Toploc lm
                                    ∧ ncpDenvalueLimit lev0 lev lk (Stack lst) (Memory lst)
                                    ∧ ncpOMemInDMem (Memory lst) lm
                                    ∧ ncpElementsNoSubPref (Memory lst) lm
                                    ∧ ncpNewSelfPoint (Memory lst) lm
∧ Address lev = Address lev0
) ∧
                  (TypeSafe lev (Accounts lst) (Stack lst) (Memory lst) (Storage lst ) lcd) ∧
                  (TypeSafe lev0 (Accounts lst) lk lm (Storage lst ) lcd0) ∧
                   fullyInitialised lev (Accounts lst) (Stack lst) ∧ fullyInitialised lev0 (Accounts
lst) lk
```

```
⟶
                    (TypeSafe ev (Accounts lst) k m (Storage lst ) cd) ∧
 fullyInitialised ev (Accounts lst) k
∧ (¬lcp ⟶ (∀ locs tp. MCon tp (Memory lst) locs ⟶ MCon tp m locs)
                                        ∧ Toploc (Memory lst) ≤ Toploc m
                                        ∧ ncpDenvalueLimit ev lev k (Stack lst) (Memory lst)
                                        ∧ ncpOMemInDMem (Memory lst) m
                                        ∧ ncpElementsNoSubPref (Memory lst) m
                                        ∧ ncpNewSelfPoint (Memory lst) m)"
"∀ v3' t3' g3'. rexp l3 ev3 cd3 st3 g3 = Normal ((v3', t3'), g3') ∧ TypeSafe ev3 (Accounts st3) (Stack
st3) (Memory st3) (Storage st3 ) cd3
                                         ∧ fullyInitialised ev3 (Accounts st3) (Stack st3)⟶
                     (case t3' of Value typ ⇒ (typeCon typ (extractValueType(v3')) ∧ (∃ xx. v3' =
KValue xx))
                                        | Calldata struct ⇒ (MCon struct cd3 (extractValueType(v3'))
∧ (∃ xx. v3' = KCDptr xx)
                                             ∧ (∃ stloc tp'' p. (Calldata tp'', Stackloc stloc) |∈|
fmran (Denvalue ev3)∧ accessStore stloc (Stack st3) =Some (KCDptr p)
                                             ∧ ((tp'' = struct ∧ v3' = KCDptr p) ∨
                                             (∃ len arr. (extractValueType v3') ≠ p ∧ tp'' = MTArray
len arr ∧ CompMemType cd3 len arr struct p (extractValueType v3'))))
                                        | type.Memory struct ⇒ (MCon struct (Memory st3)
(extractValueType(v3'))∧ (∃ xx. v3' = KMemptr xx)
                                         ∧ (∃ stloc tp'' p. (type.Memory tp'', Stackloc stloc) |∈|
fmran (Denvalue ev3)∧ accessStore stloc (Stack st3) =Some (KMemptr p)
                                             ∧ ((tp'' = struct ∧ v3' = KMemptr p) ∨
                                             (∃ len arr. (extractValueType v3') ≠ p ∧ tp'' = MTArray
len arr ∧
                                                 CompMemType (Memory st3) len arr struct p
(extractValueType v3'))))
                                        | type.Storage struct ⇒ (SCon struct (extractValueType(v3'))
(Storage st3 (Address ev3))) ∧ (∃ xx. v3' = KStoptr xx)
                                             ∧ (∃ stloc tp''. (type.Storage tp'', Storeloc stloc) |∈|
fmran (Denvalue ev3)
                                             ∧(
                                                      (((tp'' = struct ∧ v3' = KStoptr
stloc) ∨
                                                      ((extractValueType v3') ≠ stloc ∧
CompStoType tp'' struct stloc (extractValueType v3'))))
                                                   )))"
⟨proof⟩

lemmas exprTypeconInduct = exprTypeconInduct_aux[rule_format]

end
end
```

15 Supporting lemmas for statement preservation (TypeSafe_Statement_Support)

```
theory TypeSafe_Statement_Support
  imports TypeSafe_Expressions
begin

context typesafe_base
begin
```

15.1 Stack updates and value conversion

```
lemma diffTypeDiffLoc:
  assumes "st' = st(|Stack := updateStore l v (Stack st), Gas:=g|)"
    and "v' ≠ v"
    and "accessStore l1 (Stack st') = Some v'"
```

shows *"l1 ≠ l"* ⟨*proof*⟩

lemma *stackSingleUpdate:*
assumes *"st' = st(|Stack := updateStore l v (Stack st), Gas:=g|)"*
shows *"∀nl ≠l. accessStore nl (Stack st) = accessStore nl (Stack st')"*
⟨*proof*⟩

lemma *updateOneType:*
assumes *"st' = st(|Stack := updateStore l v (Stack st), Gas:=g|)"*
and *"v' ≠ v"*
shows *"accessStore l1 (Stack st') = Some v' ⟹ accessStore l1 (Stack st) = Some v'"*
⟨*proof*⟩

lemma *typeSafeLocExists:*
assumes *"TypeSafe ev (Accounts st) (Stack st) (Memory st) (Storage st) cd"*
and *"(t, l) |∈| fmran (Denvalue ev)"*
and *"l = Stackloc loc"*
and *"unique_locations (Denvalue ev)"*
and *"balanceTypes (Accounts st)"*
shows *"∃val. accessStore loc (Stack st) = Some(val)"*
⟨*proof*⟩

lemma *typeConConvert:*
assumes *"local.expr ex env cd (st) g' = Normal ((KValue v, Value t2), g)"*
and *"TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"*
and *"convert t2 t' v = Some (v)"*
and *"fullyInitialised env (Accounts st) (Stack st)"*
shows *"typeCon t' v"*
⟨*proof*⟩

15.2 Locations returned by left-hand expressions

If storage is *SCon* and *local.lexp* returns a location in it, the types agree.

lemma *lexpStorage:*
assumes *"TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"*
and *"lexp lv env cd (st(|Gas := g|)) g = Normal ((LStoreloc locChanged, type.Storage (STValue t')),g')"*
and *"fullyInitialised env (Accounts st) (Stack st)"*

obtains *(1)*
"((type.Storage (STValue t'), Storeloc locChanged) |∈| fmran (Denvalue env))"
| (2) l t
where *"((type.Storage t, Storeloc l) |∈| fmran (Denvalue env))"*
and *"(TypedStoSubpref locChanged l t ∧ CompStoType t (STValue t') l locChanged)"*

| (3) l' t l
where *"((type.Storage t, Stackloc l) |∈| fmran (Denvalue env))"*
and *"accessStore l (Stack st) = Some (KStoptr l')"*
and *"(TypedStoSubpref locChanged l' t ∧ CompStoType t (STValue t') l' locChanged)"*
⟨*proof*⟩

15.3 Storage variables of fully initialised environments

lemma *typeSafe_storeloc_scon_from_denvalue:*
assumes *ts: "TypeSafe env acc sk mem sto cd"*
and *inDen: "(type.Storage tp, Storeloc i) |∈| fmran (Denvalue env)"*
shows *"SCon tp i (sto (Address env))"*
⟨*proof*⟩

lemma *fi_contract_var_to_denvalue_storeloc:*
assumes *fi: "fullyInitialised env acc sk"*
and *addrEq: "Address e = Address env"*

```
    and typedE: "Type (acc (Address e)) = Some (atype.Contract (Contract e))"
    and epLookup: "ep $$ Contract e = Some (ct, dud)"
    and ctVar: "ct $$ i = Some (Var tp)"
  shows "Denvalue env $$ i = Some (type.Storage tp, Storeloc i)"
⟨proof⟩

lemma fiPtr_parent_from_fullyInitialised:
  assumes fi: "fullyInitialised env acc sk"
    and inDen: "(type.Storage t, Stackloc l) |∈| fmran (Denvalue env)"
    and ptr: "accessStore l sk = Some (KStoptr p)"
  shows "∃ t' l'. (type.Storage t', Storeloc l') |∈| fmran (Denvalue env) ∧ CompStoType t' t l' p"
  ⟨proof⟩

lemma fi_denvalue_storeloc_to_contract_var:
  assumes fi: "fullyInitialised env acc sk"
    and epLookup: "ep $$ Contract env = Some (ct, dud)"
    and inDen: "(type.Storage tp, Storeloc i) |∈| fmran (Denvalue env)"
  shows "ct $$ i = Some (Var tp)"
⟨proof⟩

lemma safeContract_field_scon_from_typeCompat:
  assumes ts: "TypeSafe env acc sk mem sto cd"
    and fi: "fullyInitialised env acc sk"
    and addrEq: "Address e = Address env"
    and typedE: "Type (acc (Address e)) = Some (atype.Contract (Contract e))"
    and epLookup: "ep $$ Contract e = Some (ct, dud)"
    and ctVar: "ct $$ i = Some (Var tp)"
  shows "SCon tp i (sto (Address env))"
⟨proof⟩

lemma safeContract_other_address_preserved:
  assumes sc: "safeContract acc sto"
    and stoEq: "∀ a. a ≠ a0 ⟶ sto' a = sto a"
    and addNe: "Address e ≠ a0"
    and typedE: "Type (acc (Address (e::environment))) = Some (atype.Contract (Contract e))"
    and epLookup: "ep $$ Contract e = Some (ct, dud)"
    and ctVar: "ct $$ i = Some (Var tp)"
  shows "SCon tp i (sto' (Address e))"
⟨proof⟩

lemma copy_dest_scon_from_rel:
  assumes copyStep: "copy p l x t (Storage st (Address env)) = Some s"
    and tEq: "t' = STArray x t"
    and rootSCon: "SCon (STArray x t) l (Storage st (Address env))"
    and srcSCon: "SCon (STArray x t) p (Storage st (Address env))"
    and denSub: "(if TypedStoSubpref stloc lsrc tsrc then CompStoType tsrc tp'' lsrc stloc
                 else if TypedStoSubpref lsrc stloc tp'' then CompStoType tp'' tsrc stloc lsrc else
True)"
    and paInf: "(tp'' = (STArray x t) ∧ KStoptr p = KStoptr stloc
                ∨ extractValueType (KStoptr p) ≠ stloc ∧ CompStoType tp'' (STArray x t) stloc
(extractValueType (KStoptr p)))"
    and rel: "CompStoType tsrc (STArray x t) lsrc l"
  shows "SCon (STArray x t) l s"
⟨proof⟩
```

15.4 Memory selection

```
lemma originalMConStillMCon:
  assumes "MCon struct (Memory st) x3"
    and "(∀ tloc loc. Toploc (Memory st) ≤ tloc ∧ LSubPrefL2 loc (ShowL_nat tloc) ⟶ accessStore loc
(Memory st) = None)"
    and "∀ locs. ¬ LSubPrefL2 locs (ShowL_nat (Toploc (Memory st))) ⟶ accessStore locs (Memory st) =
accessStore locs (Memory st')"
    and "∀ locs. ¬ TypedMemSubPref locs (ShowL_nat (Toploc (Memory st))) t' ⟶ accessStore
```

```
      locs (snd (allocate (Memory st))) = accessStore locs (Memory st')"
  shows " MCon struct (Memory st') x3" ⟨proof⟩


lemma mselReturnAlwaysHash:
  assumes"msel True t l'' x22 env cd st g = Normal ((l, t'), g')"
    and "TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"
    and "MCon t (Memory st) l''"
    and "fullyInitialised env (Accounts st) (Stack st)"
  shows "∃prnt i len arr len' arr'. t= MTArray len arr ∧ l = hash prnt (ShowL_nat i) ∧ arr' = t'
      ∧ i < len' ∧ MCon (MTArray len' arr') (Memory st) prnt
                        ∧ (prnt = l'' ∧ len = len' ∧ arr' = arr ∨ CompMemType (Memory st) len arr
(MTArray len' arr') l'' prnt) " ⟨proof⟩

lemma lexpIndexMem:
  assumes "lexp lv env cd st g = Normal ((LMemloc l, type.Memory t'), g')"
    and "TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"
    and "fullyInitialised env (Accounts st) (Stack st)"

obtains (1) x21 x22 tp tParent l' l'' prnt len' arr' i
where "lv = Ref x21 x22"
  and "(tp, Stackloc l') |∈| fmran (Denvalue env)"
  and "accessStore l' (Stack st) = Some (KMemptr l'')"
  and "tp = type.Memory tParent"
  and "MCon tParent (Memory st) l'''"
  and "accessTypeStore l (Memory st) = Some (t')"
  and "(∃len arr. tParent = MTArray len arr ∧
      (case t' of MTArray l' ar' ⇒
          ∃p. accessStore l (Memory st) = Some (MPointer p) ∧ CompMemType (Memory st) len arr t' l'' p
      | MTValue val ⇒ CompMemType (Memory st) len arr t' l'' l )
∧ l = hash prnt (ShowL_nat i) ∧ i < len' ∧ arr' = t' ∧ MCon (MTArray len' arr') (Memory st) prnt
    ∧ (prnt = l'' ∧ len = len' ∧ arr' = arr ∨ CompMemType (Memory st) len arr (MTArray len' arr') l''
prnt))"
⟨proof⟩

lemma lexpStorageG:
  assumes "TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"

and "lexp lv env cd (st(|Gas := g|)) g = Normal ((LStoreloc locChanged, type.Storage t'),g')"
and "fullyInitialised env (Accounts st) (Stack st)"

obtains (1)
  "((type.Storage t', Storeloc locChanged) |∈| fmran (Denvalue env))"
| (2) l t
where "((type.Storage t, Storeloc l) |∈| fmran (Denvalue env))"
  and "(TypedStoSubpref locChanged l t ∧ CompStoType t t' l locChanged)"

| (3) l' t l
where "((type.Storage t, Stackloc l) |∈| fmran (Denvalue env))"
  and "accessStore l (Stack st) = Some (KStoptr l')"
  and "(TypedStoSubpref locChanged l' t ∧ CompStoType t t' l' locChanged)"
⟨proof⟩
```

15.5 The state invariant

`StateInvariant` is the conclusion of the main theorem. The resulting state is type-safe and fully initialised, and memory only grows: existing types, pointers and values are kept, unallocated locations below the old top stay unallocated, and the top location does not decrease.

```
definition allStoresSCon::"(address ⇒ storageT) ⇒ bool"
  where "allStoresSCon stoN = (∀e ct t' l'. ep $$ Contract e = Some ct
∧ (type.Storage t', Storeloc l') |∈| fmran (Denvalue (e::environment)) ⟶ (SCon t' l' (stoN (Address
(e::environment)))))"
```

definition *StateInvariant :: "environment ⇒ state ⇒ state ⇒ calldataT ⇒ bool"* **where**
"StateInvariant ev st_orig st_curr cd ≡
TypeSafe ev (Accounts st_curr) (Stack st_curr) (Memory st_curr) (Storage st_curr) cd ∧
fullyInitialised ev (Accounts st_curr) (Stack st_curr) ∧
(∀ locs t. accessTypeStore locs (Memory st_orig) = Some t ⟶ accessTypeStore locs (Memory st_curr) = Some t) ∧
(∀ locs v. accessStore locs (Memory st_orig) = Some (MPointer v) ⟶ (∃ v'. accessStore locs (Memory st_curr) = Some (MPointer v'))) ∧
(∀ locs v. accessStore locs (Memory st_orig) = Some (MValue v) ⟶ (∃ v'. accessStore locs (Memory st_curr) = Some (MValue v'))) ∧
(∀ i loc. i < Toploc (Memory st_orig) ∧ LSubPrefL2 loc (ShowL$_{nat}$ i) ⟶
accessStore loc (Memory st_orig) = None ⟶ accessStore loc (Memory st_curr) = None) ∧
Toploc (Memory st_orig) ≤ Toploc (Memory st_curr)
"

15.6 Memory-to-memory copies

lemma *cpm2m_subPrefixPersist:*
assumes *"subPrefixStructuralConsistency mo"*
and *"accessStore (ShowL$_{nat}$ (Toploc mo)) mo = None"*
and *"SomeValSomeTyp mo"*
and *sameTypeAccess:"∀ locs. ¬ LSubPrefL2 locs (ShowL$_{nat}$ (Toploc mo)) ∨ locs = (ShowL$_{nat}$ (Toploc mo)) ⟶*
accessTypeStore locs (mo) = accessTypeStore locs m"
and *lessThan:"lessThanTopLocs mo"*
and *sameTypeAccess2:" ∀ locs. ¬ TypedMemSubPref locs (ShowL$_{nat}$ (Toploc mo)) (MTArray x t) ∨ locs = (ShowL$_{nat}$ (Toploc mo)) ⟶*
accessTypeStore locs (mo) = accessTypeStore locs m"
and *tps:"∀ destl'.*
TypedMemSubPref destl' (ShowL$_{nat}$ (Toploc mo)) (MTArray x t) ⟶
(∃ stt. CompMemType m x t stt (ShowL$_{nat}$ (Toploc mo)) destl' ∧
(case stt of MTArray parent_len parent_arr ⇒ ∀ i<parent_len. accessTypeStore (hash destl' (ShowL$_{nat}$ i)) m = Some parent_arr
| MTValue pval ⇒ accessTypeStore destl' m = Some (MTValue pval)))"
and *MCondest:" MCon (MTArray x t) m (ShowL$_{nat}$ (Toploc (mo)))"*
and *selfPoint:"∀ l l'. TypedMemSubPref l (ShowL$_{nat}$ (Toploc mo)) (MTArray x t) ∧ accessStore l m = Some (MPointer l') ⟶ l' = l"*
and *tps2:"∀ i<x. accessTypeStore (hash (ShowL$_{nat}$ (Toploc mo)) (ShowL$_{nat}$ i)) m = Some t"*
and *nonLocChanged:"∀ locs. ¬ LSubPrefL2 locs (ShowL$_{nat}$ (Toploc mo)) ∨ locs = (ShowL$_{nat}$ (Toploc mo)) ⟶ accessStore locs (mo) = accessStore locs m"*
and *a32:"∀ locs. ¬ TypedMemSubPref locs (ShowL$_{nat}$ (Toploc mo)) (MTArray x t) ⟶ accessStore locs mo = accessStore locs m"*
shows *"subPrefixStructuralConsistency (m)"* ⟨*proof*⟩

lemma *cpm2m_singleLChange:*
assumes *"MCon struct m x3"*
assumes *structss:"(case struct of*
MTArray len arr ⇒
(∀ i<len. accessTypeStore (hash x3 (ShowL$_{nat}$ i)) m = Some arr)
| MTValue val ⇒ accessTypeStore x3 m = Some (MTValue val))"
and *"updateStore l (MPointer tloc) m = m'"*
and *lOld2:"∃ p. accessStore l m = Some (MPointer p)"*
and *accl:"accessTypeStore l (m) = Some (MTArray x t)"*
and *MCondest2:"MCon (MTArray x t) m' (tloc)"*
and *subPrefM:"subPrefixStructuralConsistency (m)"*
and *NonChangeM'm:"∀ locs. locs ≠ l ⟶ accessStore locs m = accessStore locs m'"*
shows *"MCon struct m' x3"*
⟨*proof*⟩

end

end

16 Preservation for assignments (TypeSafe_Assign)

theory *TypeSafe_Assign*
  imports *TypeSafe_Statement_Support*
begin

context *typesafe_base*
begin

Each lemma covers one case of the parent's *assign* rule for *ASSIGN*, distinguished by the kinds of the assigned value and of the target location. Together they give the *ASSIGN* case of *TypeSafe_Statements*.

Assignment case 1: a value assigned to a stack variable.

lemma *TypeSafe_Assign_1:*
  assumes *"TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"*
    and *"stmt (ASSIGN lv ex) env cd st = Normal ((), st')"*
    and *"fullyInitialised env (Accounts st) (Stack st)"*
    and *"expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st - costs (ASSIGN lv ex) env cd st) = Normal ((KValue v, Value t2), g)"*
    and *"lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LStackloc l2, Value t'),g')"*
    and *"convert t2 t' v = Some v'"*
    and *"st' = st⦇state.Gas := g', Stack := updateStore l2 (KValue v') (Stack st)⦈"*
  shows *"StateInvariant env st st' cd"*
⟨*proof*⟩

Assignment case 2: a value assigned to a storage location.

lemma *TypeSafe_Assign_2:*
  assumes *"TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"*
    and *"stmt (ASSIGN lv ex) env cd st = Normal ((), st')"*
    and *"fullyInitialised env (Accounts st) (Stack st)"*
    and *"expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st - costs (ASSIGN lv ex) env cd st) = Normal ((KValue v, Value t2), g)"*
    and *"lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LStoreloc locationChanged, type.Storage (STValue t')),g')"*
    and *"convert t2 t' v = Some v'"*
    and *"st' = st⦇state.Gas := g', Storage := (Storage st) (Address env := (fmupd locationChanged v' (Storage st (Address env))))⦈"*
  shows *"StateInvariant env st st' cd"*
⟨*proof*⟩

Assignment case 3: a value assigned to a memory location.

lemma *TypeSafe_Assign_3:*
  assumes *"TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"*
    and *"stmt (ASSIGN lv ex) env cd st = Normal ((), st')"*
    and *"fullyInitialised env (Accounts st) (Stack st)"*
    and *"expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st - costs (ASSIGN lv ex) env cd st) = Normal ((KValue v, Value t), g)"*
    and *"lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LMemloc l, type.Memory (MTValue t')),g')"*
    and *"convert t t' v = Some v'"*
    and *"st' = st⦇state.Gas := g', Memory := updateStore l (MValue v') (Memory st)⦈"*
  shows *"StateInvariant env st st' cd"*
⟨*proof*⟩

Assignment case 4: a calldata array assigned to a memory variable on the stack: the array is copied into fresh memory.

lemma *TypeSafe_Assign_4:*
  assumes *"TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"*
    and *"stmt (ASSIGN lv ex) env cd st = Normal ((), st')"*
    and *"fullyInitialised env (Accounts st) (Stack st)"*
    and *"expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st - costs (ASSIGN lv
        ex) env cd st) = Normal ((KCDptr p, Calldata (MTArray x t)), g)"*
    and *"lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LStackloc l, type.Memory t'),g')"*
    and *"(MTArray x t) =t'"*

```
    and "accessStore l (Stack st) = Some (KMemptr p')"
    and "(updateStore l (KMemptr (ShowL_nat (Toploc (Memory st)))) (Stack st)) = st''"
    and "cpm2m p (ShowL_nat (Toploc (Memory st))) x t cd (snd (allocate (Memory st))) = Some m"
    and "st' = st⦇state.Gas := g', Memory := m, Stack := st''⦈"
  shows "StateInvariant env st st' cd"
⟨proof⟩
```

Assignment case 5: a calldata array assigned to a storage variable reached through a stack pointer: the array is copied into storage.

```
lemma TypeSafe_Assign_5:
  assumes "TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"
    and "stmt (ASSIGN lv ex) env cd st = Normal ((), st')"
    and "fullyInitialised env (Accounts st) (Stack st)"
    and "expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st
- costs (ASSIGN lv
        ex) env cd st) = Normal ((KCDptr p, Calldata (MTArray x t)), g)"
    and "lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LStackloc l, type.Storage t'),g')"
    and "accessStore l (Stack st) = Some (KStoptr p')"
    and "cps2mTypeCompatible t' (MTArray x t) "
    and "cpm2s p p' x t cd (Storage st (Address env)) = Some s"
    and "st' = st⦇state.Gas := g', Storage := (Storage st) (Address env := s)⦈"
  shows "StateInvariant env st st' cd"
⟨proof⟩
```

Assignment case 6: a calldata array assigned to a storage location: the array is copied into storage.

```
lemma TypeSafe_Assign_6:
  assumes "TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"
    and "stmt (ASSIGN lv ex) env cd st = Normal ((), st')"
    and "fullyInitialised env (Accounts st) (Stack st)"
    and "expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st
- costs (ASSIGN lv
        ex) env cd st) = Normal ((KCDptr p, Calldata (MTArray x t)), g)"
    and "lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LStoreloc l, type.Storage t'),g')"
    and "cps2mTypeCompatible t' (MTArray x t)"
    and "cpm2s p l x t cd (Storage st (Address env)) = Some s"
    and "st' = st⦇state.Gas := g', Storage := (Storage st) (Address env := s)⦈"
  shows "StateInvariant env st st' cd"
⟨proof⟩
```

Assignment case 7: a calldata array assigned to a memory location: the array is copied into fresh memory and the location is updated to point to it.

```
lemma TypeSafe_Assign_7:
  assumes "TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"
    and "stmt (ASSIGN lv ex) env cd st = Normal ((), st')"
    and "fullyInitialised env (Accounts st) (Stack st)"
    and "expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st
- costs (ASSIGN lv
        ex) env cd st) = Normal ((KCDptr p, Calldata (MTArray x t)), g)"
    and "lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LMemloc l, type.Memory t'),g')"
    and "(MTArray x t) = t'"
    and "cpm2m p (ShowL_nat (Toploc (Memory st))) x t cd (snd (allocate (Memory st))) = Some m"
    and "(updateStore l (MPointer (ShowL_nat (Toploc (Memory st)))) m) = m'"
    and "st' = st⦇state.Gas := g', Memory := m'⦈"
  shows "StateInvariant env st st' cd"
⟨proof⟩
```

Assignment case 8: a memory array assigned to a memory variable on the stack: the stack now points to the same array.

```
lemma TypeSafe_Assign_8:
  assumes "TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"
    and "stmt (ASSIGN lv ex) env cd st = Normal ((), st')"
    and "fullyInitialised env (Accounts st) (Stack st)"
    and "expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st
- costs (ASSIGN lv
```

```
        ex) env cd st) = Normal ((KMemptr p, type.Memory (MTArray x t)), g)"
    and "lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LStackloc l, type.Memory t'),g')"
    and "t' = MTArray x t"
    and "st' = st⦇state.Gas := g', Stack := updateStore l (KMemptr p) (Stack st)⦈"
  shows "StateInvariant env st st' cd"
```
⟨*proof*⟩

Assignment case 9: a memory array assigned to a storage variable reached through a stack pointer: the array is copied into storage.

```
lemma TypeSafe_Assign_9:
  assumes "TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"
    and "stmt (ASSIGN lv ex) env cd st = Normal ((), st')"
    and "fullyInitialised env (Accounts st) (Stack st)"
    and "expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st
- costs (ASSIGN lv
        ex) env cd st) = Normal ((KMemptr p, type.Memory (MTArray x t)), g)"
    and "lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LStackloc l, type.Storage t'),g')"
    and "cps2mTypeCompatible t' (MTArray x t)"
    and "accessStore l (Stack st) = Some (KStoptr p')"
    and "cpm2s p p' x t (Memory st) (Storage st (Address env)) = Some s"
    and "st' = st⦇state.Gas := g', Storage := (Storage st) (Address env := s)⦈"
  shows "StateInvariant env st st' cd"
```
⟨*proof*⟩

Assignment case 10: a memory array assigned to a storage location: the array is copied into storage.

```
lemma TypeSafe_Assign_10:
  assumes "TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"
    and "stmt (ASSIGN lv ex) env cd st = Normal ((), st')"
    and "fullyInitialised env (Accounts st) (Stack st)"
    and "expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st
- costs (ASSIGN lv
        ex) env cd st) = Normal ((KMemptr p, type.Memory (MTArray x t)), g)"
    and "lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LStoreloc l, type.Storage t'),g')"
    and "cps2mTypeCompatible t' (MTArray x t)"
    and "cpm2s p l x t (Memory st) (Storage st (Address env)) = Some s"
    and "st' = st⦇state.Gas := g', Storage := (Storage st) (Address env := s)⦈"
  shows "StateInvariant env st st' cd"
```
⟨*proof*⟩

Assignment case 11: a memory array assigned to a memory location: the location is updated to point to the same array.

```
lemma TypeSafe_Assign_11:
  assumes "TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"
    and "stmt (ASSIGN lv ex) env cd st = Normal ((), st')"
    and "fullyInitialised env (Accounts st) (Stack st)"
    and "expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st
- costs (ASSIGN lv
        ex) env cd st) = Normal ((KMemptr p, type.Memory (MTArray x t)), g)"
    and "lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LMemloc l, type.Memory t'),g')"
    and "t' = MTArray x t"
    and "st' = st⦇state.Gas := g', Memory := updateStore l (MPointer p) (Memory st)⦈"
  shows "StateInvariant env st st' cd"
```
⟨*proof*⟩

Assignment case 12: a storage array assigned to a memory variable on the stack: the array is copied into fresh memory.

```
lemma TypeSafe_Assign_12:
  assumes "TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"
    and "stmt (ASSIGN lv ex) env cd st = Normal ((), st')"
    and "fullyInitialised env (Accounts st) (Stack st)"
    and "expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st
- costs (ASSIGN lv
        ex) env cd st) = Normal ((KStoptr p, type.Storage (STArray x t)), g)"
```

and *"lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LStackloc l, type.Memory t'),g')"*
and *"cps2mTypeCompatible (STArray x t) t'"*
and *"accessStore l (Stack st) = Some (KMemptr p')"*
and *"st'' = updateStore l (KMemptr (ShowL$_{nat}$ (Toploc (Memory st)))) (Stack st)"*
and *"cps2m p (ShowL$_{nat}$ (Toploc (Memory st))) x t (Storage st (Address env)) (snd (allocate(Memory st))) = Some m"*
and *"st' = st⦇state.Gas := g', Memory := m, Stack := st''⦈"*
shows *"StateInvariant env st st' cd"*
⟨*proof*⟩

Assignment case 13: a storage array assigned to a storage pointer on the stack: the stack now points to the same storage array.

lemma *TypeSafe_Assign_13:*
assumes *"TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"*
and *"stmt (ASSIGN lv ex) env cd st = Normal ((), st')"*
and *"fullyInitialised env (Accounts st) (Stack st)"*
and *"expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st - costs (ASSIGN lv ex) env cd st) = Normal ((KStoptr p, type.Storage (STArray x t)), g)"*
and *"lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LStackloc l, type.Storage t'),g')"*
and *"t' = STArray x t"*
and *"st' = st⦇state.Gas := g', Stack := updateStore l (KStoptr p) (Stack st)⦈"*
shows *"StateInvariant env st st' cd"*
⟨*proof*⟩

Assignment case 14: a storage array assigned to a storage location: the array is copied within storage.

lemma *TypeSafe_Assign_14:*
assumes *"TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"*
and *"stmt (ASSIGN lv ex) env cd st = Normal ((), st')"*
and *"fullyInitialised env (Accounts st) (Stack st)"*
and *"expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st - costs (ASSIGN lv ex) env cd st) = Normal ((KStoptr p, type.Storage (STArray x t)), g)"*
and *"lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LStoreloc l, type.Storage t'),g')"*
and *"t' = (STArray x t)"*
and *"copy p l x t (Storage st (Address env)) = Some s"*
and *"st' = st⦇state.Gas := g', Storage := (Storage st) (Address env := s)⦈"*
shows *"StateInvariant env st st' cd"*
⟨*proof*⟩

Assignment case 15: a storage array assigned to a memory location: the array is copied into fresh memory and the location is updated to point to it.

lemma *TypeSafe_Assign_15:*
assumes *"TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"*
and *"stmt (ASSIGN lv ex) env cd st = Normal ((), st')"*
and *"fullyInitialised env (Accounts st) (Stack st)"*
and *"expr ex env cd (st⦇state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st⦈) (state.Gas st - costs (ASSIGN lv ex) env cd st) = Normal ((KStoptr p, type.Storage (STArray x t)), g)"*
and *"lexp lv env cd (st⦇state.Gas := g⦈) g = Normal((LMemloc l, type.Memory t'),g')"*
and *"cps2mTypeCompatible (STArray x t) t'"*
and *"cps2m p (ShowL$_{nat}$ (Toploc (Memory st))) x t (Storage st (Address env)) (snd (allocate(Memory st))) = Some m"*
and *"updateStore l (MPointer ((ShowL$_{nat}$ (Toploc (Memory st))))) m = m'"*
and *"st' = st⦇state.Gas := g', Memory := m'⦈"*
shows *"StateInvariant env st st' cd"*
⟨*proof*⟩

Assignment case 16: a storage mapping assigned to a storage pointer on the stack: the stack now points to the same mapping.

lemma *TypeSafe_Assign_16:*
assumes *"TypeSafe env (Accounts st) (Stack st) (Memory st) (Storage st) cd"*
and *"stmt (ASSIGN lv ex) env cd st = Normal ((), st')"*

```
    and "fullyInitialised env (Accounts st) (Stack st)"
    and "expr ex env cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) env cd st|)) (state.Gas st
- costs (ASSIGN lv
        ex) env cd st) = Normal ((KStoptr p, type.Storage (STMap t g')), g)"
    and "lexp lv env cd (st(|state.Gas := g|)) g = Normal((LStackloc l, type.Storage t'),t'')"
    and "t' = (STMap t g')"
    and "st' = st(|state.Gas := t'', Stack := updateStore l (KStoptr p) (Stack st)|)"
  shows "StateInvariant env st st' cd"
⟨proof⟩
```

end

end

17 Statement preservation (TypeSafe)

theory *TypeSafe*
  imports *TypeSafe_Assign*
begin

context *typesafe_base*
begin

Successful statement execution preserves the state invariant when the initial state is type-consistent and the environment is fully initialised. The assumptions of locale *typesafe_base* remain explicit prerequisites.

The proof is by induction over the statement semantics *local.stmt*, with one case per statement form. Each case shows that the resulting state is again type-safe, that the environment stays fully initialised, and that memory only grows (see *StateInvariant*).

- *SKIP* is immediate; *COMP*, *ITE* and *WHILE* follow from the induction hypotheses.
- The sixteen forms of *ASSIGN* are proved separately in *TypeSafe_Assign*.
- *INVOKE*, *EXTERNAL* and *NEW* run a body in a freshly initialised environment and use the initialisation results of *TypeSafe_Environment* with the induction hypothesis.
- *TRANSFER* uses the balance lemmas of *TypeSafe_Base_Types*.
- The two forms of *BLOCK* use the declaration results of *TypeSafe_Decl*.

```
theorem TypeSafe_Statements:
  assumes "TypeSafe ev (Accounts st) (Stack st) (Memory st) (Storage st) cd"
    and normal:"stmt smt ev cd st = Normal((),st')"
    and inits:"fullyInitialised ev (Accounts st) (Stack st)"
  shows "StateInvariant ev st st' cd"
  ⟨proof⟩
```

end

end

18 Examples (TypeSafe_Examples)

theory *TypeSafe_Examples*
  imports *TypeSafe*
begin

18.1 Satisfiability of the locale assumptions

The assumptions of *typesafe_base* hold for the empty procedure environment and for a non-empty one.

global_interpretation *typesafe_vacuous: typesafe_base costs_ex fmempty costs_min*

⟨*proof*⟩

Non-vacuous sanity check: a concrete non-empty procedure environment.

definition *nv_ct :: "(Identifier, member) fmap"*
where *"nv_ct =*
fmap_of_list
[(STR ''x'', Var (STValue TBool)),
(STR ''y'', Var (STValue (TUInt b8)))]"

definition *nv_ctor :: "(Identifier × type) list × s"*
where *"nv_ctor = ([], SKIP)"*

definition *nv_contract :: contract*
where *"nv_contract = (nv_ct, nv_ctor, SKIP)"*

definition *ep_nonvacuous :: environment$_p$*
where *"ep_nonvacuous = fmap_of_list [(STR ''C'', nv_contract)]"*

lemma *ep_nonvacuous_has_two_vars:*
"∃ct dud t1 t2.
ep_nonvacuous $$ (STR ''C'') = Some (ct, dud) ∧
ct $$ (STR ''x'') = Some (Var t1) ∧
ct $$ (STR ''y'') = Some (Var t2) ∧
(STR ''x'') ≠ (STR ''y'')"
⟨*proof*⟩

global_interpretation *typesafe_nonvacuous: typesafe_base costs_ex ep_nonvacuous costs_min*
⟨*proof*⟩

18.2 Memory traversal example

abbreviation *mymemory22::memoryT*
where *"mymemory22 ≡*
⦇Mapping = fmap_of_list
[(STR ''1.1.0'', MValue STR ''False''),
(STR ''0.1.0'', MValue STR ''True''),
(STR ''1.0'', MPointer STR ''2.0''),
(STR ''1.0.0'', MValue STR ''False''),
(STR ''0.2.0'', MValue STR ''True''),
(STR ''0.0'', MPointer STR ''0.0'')],
Toploc = 1,
Typed_Mapping = fmap_of_list
[(STR ''1.1.0'', MTValue TBool),
(STR ''0.1.0'', MTValue TBool),
(STR ''1.0'', MTArray 2 (MTValue TBool)),
(STR ''1.0.0'', MTValue TBool),
(STR ''0.2.0'', MTValue TBool),
(STR ''0.0'', MTArray 2 (MTValue TBool))]⦈"
lemma *mymemory22_locations:*
"GetAllMemoryLocations (MTArray 3 (MTArray 2 (MTValue TBool))) STR ''0'' mymemory22 =
{STR ''0.2.0'', STR ''1.0'', STR ''1.0.0'', STR ''0.0''}"
⟨*proof*⟩

18.3 Concrete preservation examples

Concrete usability demo for *typesafe_base.TypeSafe_Statements* with a non-empty contract environment.
The setup models the following Solidity contract:

```
contract C {
    bool x;
    uint8 y;

    function ping() public {
        // SKIP
```

```
    }
}
```

- contract identifier *STR ''C''* is looked up in *ep_nonvacuous*;
- storage variables *STR ''x''* and *STR ''y''* map to *Var (STValue TBool)* and *Var (STValue (TUInt b8))* in *nv_ct*;
- the environment is produced by *ffold_init*, so its *Denvalue* has *Storeloc* entries;
- statement *SKIP* plays the role of an empty function body.

definition *demo_addr :: address*
  where *"demo_addr = createAddress (STR ''A'')"*

definition *demo_sender :: address*
  where *"demo_sender = createAddress (STR ''S'')"*

definition *demo_svalue :: valuetype*
  where *"demo_svalue = createUInt b256 0"*

definition *demo_env :: environment*
  where *"demo_env =*
    ffold_init nv_ct (emptyEnv demo_addr (STR ''C'') demo_sender demo_svalue) (fmdom nv_ct)"

definition *demo_acc :: accounts*
  where *"demo_acc =*
    emptyAccount(demo_addr := emptyAcc(|Type := Some (atype.Contract (STR ''C''))|))"

definition *demo_state :: state*
  where *"demo_state =*
    (|Accounts = demo_acc,
     Stack = emptyStore,
     Memory = emptyTypedStore,
     Storage = emptyStorage,
     Gas = 1|)"

definition *demo_cd :: calldataT*
  where *"demo_cd = emptyTypedStore"*

lemma *demo_safeContract_base:*
  "typesafe_base.safeContract ep_nonvacuous demo_acc emptyStorage"
⟨*proof*⟩

lemma *demo_safeContract:*
  "typesafe_nonvacuous.safeContract demo_acc emptyStorage"
  ⟨*proof*⟩

lemma *demo_addressFormat_addr: "addressFormat demo_addr"*
  ⟨*proof*⟩

lemma *demo_addressFormat_sender: "addressFormat demo_sender"*
  ⟨*proof*⟩

lemma *demo_balanceTypes: "balanceTypes demo_acc"*
  ⟨*proof*⟩

lemma *demo_AddressTypes: "typesafe_base.AddressTypes ep_nonvacuous demo_acc"*
⟨*proof*⟩

lemma *demo_typeSafe:*
  "typesafe_base.TypeSafe ep_nonvacuous demo_env (Accounts demo_state) (Stack demo_state) (Memory demo_state) (Storage demo_state) demo_cd"
⟨*proof*⟩

lemma *demo_fullyInitialised:*
"typesafe_base.fullyInitialised ep_nonvacuous demo_env (Accounts demo_state) (Stack demo_state)"
⟨*proof*⟩

Concrete non-vacuous witness for the pointer conjunct in *typesafe_base.fullyInitialised_def*. Unlike *demo_fullyInitialised*, this setup includes a storage-typed stack binding that points to an existing storage location, so the pointer implication is exercised with a true antecedent.

definition *demo_ptr_id :: Identifier*
where *"demo_ptr_id = STR ''ptr_tmp''"*

definition *demo_ptr_stack_loc :: location*
where *"demo_ptr_stack_loc = STR ''0''"*

definition *demo_ptr_target_loc :: location*
where *"demo_ptr_target_loc = STR ''x''"*

definition *demo_env_with_stptr :: environment*
where *"demo_env_with_stptr =*
demo_env(|Denvalue := fmupd demo_ptr_id
(type.Storage (STValue TBool), Stackloc demo_ptr_stack_loc) (Denvalue demo_env)|)"

definition *demo_state_with_stptr :: state*
where *"demo_state_with_stptr =*
demo_state(|Stack := updateStore demo_ptr_stack_loc (KStoptr demo_ptr_target_loc) (Stack demo_state)|)"

lemma *demo_pointer_clause_nonvacuous:*
"∃t l p.
(type.Storage t, Stackloc l) |∈| fmran (Denvalue demo_env_with_stptr) ∧
accessStore l (Stack demo_state_with_stptr) = Some (KStoptr p) ∧
(∃t' l'. (type.Storage t', Storeloc l') |∈| fmran (Denvalue demo_env_with_stptr) ∧ CompStoType t' t l' p)"
⟨*proof*⟩

Concrete 3D memory-array witness aligned with *decl* initialisation: an outer array of length 2 whose element type is a 2D bool array.

definition *demo_mem3d_elem_type :: mtypes*
where *"demo_mem3d_elem_type = MTArray 2 (MTArray 2 (MTValue TBool))"*

definition *demo_mem3d_type :: mtypes*
where *"demo_mem3d_type = MTArray 2 demo_mem3d_elem_type"*

definition *demo_mem3d_root :: location*
where *"demo_mem3d_root = ShowL$_{nat}$ (Toploc (Memory demo_state))"*

definition *demo_mem3d :: memoryT*
where *"demo_mem3d = minit 2 demo_mem3d_elem_type (Memory demo_state)"*

definition *demo_mem3d_id :: Identifier*
where *"demo_mem3d_id = STR ''m3d''"*

definition *demo_mem3d_stack_loc :: location*
where *"demo_mem3d_stack_loc = ShowL$_{nat}$ (Toploc (Stack demo_state))"*

definition *demo_stack_mem3d :: stack*
where *"demo_stack_mem3d = push (KMemptr demo_mem3d_root) (Stack demo_state)"*

definition *demo_env_mem3d :: environment*
where *"demo_env_mem3d =*
updateEnv demo_mem3d_id (type.Memory demo_mem3d_type) (Stackloc demo_mem3d_stack_loc) demo_env"

definition *demo_state_mem3d :: state*
where *"demo_state_mem3d = demo_state(|Stack := demo_stack_mem3d, Memory := demo_mem3d|)"*

lemma *demo_mem3d_id_fresh:*
```
  "Denvalue demo_env $$ demo_mem3d_id = None"
```
⟨*proof*⟩

lemma *demo_decl_mem3d_matches:*
```
  "decl demo_mem3d_id (type.Memory demo_mem3d_type) None False
      demo_cd (Memory demo_state) (Storage demo_state (Address demo_env))
      (demo_cd, Memory demo_state, Stack demo_state, demo_env) =
   Some (demo_cd, demo_mem3d, demo_stack_mem3d, demo_env_mem3d)"
```
⟨*proof*⟩

lemma *demo_mem3d_root_empty:*
```
  "accessStore demo_mem3d_root demo_mem3d = None"
```
⟨*proof*⟩

lemma *demo_stack_mem3d_pointer:*
```
  "accessStore demo_mem3d_stack_loc demo_stack_mem3d = Some (KMemptr demo_mem3d_root)"
```
⟨*proof*⟩

lemma *demo_env_mem3d_binding:*
```
  "Denvalue demo_env_mem3d $$ demo_mem3d_id =
    Some (type.Memory demo_mem3d_type, Stackloc demo_mem3d_stack_loc)"
```
⟨*proof*⟩

lemma *demo_typeSafe_mem3d:*
```
  "typesafe_base.TypeSafe ep_nonvacuous demo_env_mem3d
    (Accounts demo_state_mem3d) (Stack demo_state_mem3d) (Memory demo_state_mem3d) (Storage
demo_state_mem3d) demo_cd"
```
⟨*proof*⟩

lemma *demo_mem3d_lessThanTopLocs:*
```
  "lessThanTopLocs demo_mem3d"
```
⟨*proof*⟩

lemma *demo_mem3d_subPrefixStructuralConsistency:*
```
  "typesafe_nonvacuous.subPrefixStructuralConsistency demo_mem3d"
```
⟨*proof*⟩

lemma *demo_mem3d_SomeValSomeTyp:*
```
  "typesafe_nonvacuous.SomeValSomeTyp demo_mem3d"
```
⟨*proof*⟩

lemma *demo_fullyInitialised_mem3d:*
```
  "typesafe_base.fullyInitialised ep_nonvacuous demo_env_mem3d (Accounts demo_state_mem3d) (Stack
demo_state_mem3d)"
```
⟨*proof*⟩

lemma *demo_stmt_normal:*
```
  "typesafe_nonvacuous.stmt SKIP demo_env demo_cd demo_state = Normal ((), demo_state)"
```
⟨*proof*⟩

lemma *demo_stmt_normal_mem3d:*
```
  "typesafe_nonvacuous.stmt SKIP demo_env_mem3d demo_cd demo_state_mem3d = Normal ((),
demo_state_mem3d)"
```
⟨*proof*⟩

Applying *typesafe_base.TypeSafe_Statements* to both concrete states.

lemma *demo_StateInvariant_from_TypeSafe_Statements:*
```
  "typesafe_base.StateInvariant ep_nonvacuous demo_env demo_state demo_state demo_cd"
```
⟨*proof*⟩

lemma *demo_mem3d_StateInvariant_from_TypeSafe_Statements:*

```
  "typesafe_base.StateInvariant ep_nonvacuous demo_env_mem3d demo_state_mem3d demo_state_mem3d demo_cd"
  ⟨proof⟩
```

Evaluating the memory example: the stack holds a reference to the outer array at location `0`, whose first element is a self-pointing inner array.

```
lemma demo_mem3d_eval:
  "demo_mem3d_root = STR ''0''"
  "accessStore demo_mem3d_stack_loc (Stack demo_state_mem3d) = Some (KMemptr (STR ''0''))"
  "accessStore (hash demo_mem3d_root (STR ''0'')) (Memory demo_state_mem3d) =
     Some (MPointer (STR ''0.0''))"
  ⟨proof⟩

end
```

References

[1] Diego Marmsoler and Achim D. Brucker. Isabelle/Solidity: A deep embedding of Solidity in Isabelle/HOL. *Archive of Formal Proofs*, July 2022. https://isa-afp.org/entries/Solidity.html, Formal proof development.

[2] Billy Thornton and Diego Marmsoler. Type safety for Isabelle/Solidity. In Chutiporn Anutariya and Marcello M. Bonsangue, editors, *Theoretical Aspects of Computing - ICTAC 2024 - 21st International Colloquium, Bangkok, Thailand, November 25-29, 2024, Proceedings*, volume 15373 of *Lecture Notes in Computer Science*, pages 313–330. Springer, 2024.

[3] Billy Thornton and Diego Marmsoler. Type safety for Isabelle/Solidity. *Theoretical Computer Science*, 1067:115632, 2026.