

Formalization of CommCSL: A Relational Concurrent Separation Logic for Proving Information Flow Security in Concurrent Programs

Thibault Dardinier
Department of Computer Science
ETH Zurich, Switzerland

February 6, 2026

Abstract

Information flow security ensures that the secret data manipulated by a program does not influence its observable output. Proving information flow security is especially challenging for concurrent programs, where operations on secret data may influence the execution time of a thread and, thereby, the interleaving between threads. Such internal timing channels may affect the observable outcome of a program even if an attacker does not observe execution times. Existing verification techniques for information flow security in concurrent programs attempt to prove that secret data does not influence the relative timing of threads. However, these techniques are often restrictive (for instance because they disallow branching on secret data) and make strong assumptions about the execution platform (ignoring caching, processor instructions with data-dependent execution time, and other common features that affect execution time).

In this entry, we formalize and prove the soundness of `COMM-CSL` [1], a novel relational concurrent separation logic for proving secure information flow in concurrent programs that lifts these restrictions and does not make any assumptions about timing behavior. The key idea is to prove that all mutating operations performed on shared data commute, such that different thread interleavings do not influence its final value. Crucially, commutativity is required only for an abstraction of the shared data that contains the information that will be leaked to a public output. Abstract commutativity is satisfied by many more operations than standard commutativity, which makes our technique widely applicable.

Contents

1	State Model	3
1.1	Partial Heaps	3
1.2	Fractional Permissions	4
1.3	Permission Heaps	8
1.4	Extended Heaps	10
2	Imperative Concurrent Language	17
2.1	Language Syntax and Semantics	17
2.1.1	Semantics of expressions	18
2.1.2	Semantics of commands	18
2.1.3	Abort semantics	19
2.2	Useful Definitions and Results	20
3	CommCSL	24
3.1	Assertion Language	25
3.2	Rules of the Logic	33
4	Soundness of CommCSL	35
4.1	Abstract Commutativity	35
4.2	Consistency	42
4.3	Safety and Hoare Triples	49
4.3.1	Preliminaries	49
4.3.2	Safety	51
4.3.3	Useful results about safety	55
4.3.4	Hoare triples	56
4.4	Soundness of the Rules	57
4.4.1	Skip	58
4.4.2	Assign	58
4.4.3	Alloc	58
4.4.4	Write	59
4.4.5	Read	60
4.4.6	Share	60
4.4.7	Atomic	62
4.4.8	Parallel	63
4.4.9	If	65
4.4.10	Sequential composition	66
4.4.11	Frame rule	67
4.4.12	Consequence	68
4.4.13	Existential	68
4.4.14	While loops	68
4.4.15	CommCSL is sound	72
4.5	Corollaries	72

1 State Model

1.1 Partial Heaps

In this file, we prove useful lemmas about partial maps. Partial maps are used to define permission heaps (see `FractionalHeap.thy`) and the family of unique action guard states (see `StateModel.thy`).

theory *PartialMap*

imports *Main*

begin

type-synonym (*'a*, *'b*) *map* = *'a* \rightarrow *'b*

fun *compatible-options* :: (*'a* \Rightarrow *'a* \Rightarrow *bool*) \Rightarrow *'a option* \Rightarrow *'a option* \Rightarrow *bool* **where**
 compatible-options *f* (*Some a*) (*Some b*) \longleftrightarrow *f a b*
| *compatible-options* - - - \longleftrightarrow *True*

fun *merge-option* :: (*'b* \Rightarrow *'b* \Rightarrow *'b*) \Rightarrow *'b option* \Rightarrow *'b option* \Rightarrow *'b option* **where**
 merge-option - *None None* = *None*
| *merge-option* - (*Some a*) *None* = *Some a*
| *merge-option* - *None (Some b)* = *Some b*
| *merge-option* *f (Some a) (Some b)* = *Some (f a b)*

definition *merge-options* :: (*'c* \Rightarrow *'c* \Rightarrow *'c*) \Rightarrow (*'b*, *'c*) *map* \Rightarrow (*'b*, *'c*) *map* \Rightarrow (*'b*, *'c*) *map* **where**
 merge-options *f a b p* = *merge-option* *f (a p) (b p)*

Two maps are compatible iff they are compatible pointwise (i.e., if both define values, then those values are compatible)

definition *compatible-maps* :: (*'b* \Rightarrow *'b* \Rightarrow *bool*) \Rightarrow (*'a*, *'b*) *map* \Rightarrow (*'a*, *'b*) *map* \Rightarrow *bool* **where**
 compatible-maps *f h1 h2* \longleftrightarrow (\forall *hl*. *compatible-options* *f (h1 hl) (h2 hl)*)

lemma *compatible-mapsI*:

assumes $\bigwedge x a b. h1 x = \text{Some } a \wedge h2 x = \text{Some } b \implies f a b$

shows *compatible-maps* *f h1 h2*

<proof>

definition *map-included* :: (*'a*, *'b*) *map* \Rightarrow (*'a*, *'b*) *map* \Rightarrow *bool* **where**

map-included *h1 h2* \longleftrightarrow ($\forall x. h1 x \neq \text{None} \longrightarrow h1 x = h2 x$)

lemma *map-includedI*:

assumes $\bigwedge x r. h1 x = \text{Some } r \implies h2 x = \text{Some } r$

shows *map-included* *h1 h2*

<proof>

lemma *compatible-maps-empty*:

compatible-maps *f h (Map.empty)*

<proof>

lemma *compatible-maps-comm*:
 compatible-maps (=) *h1 h2* \longleftrightarrow *compatible-maps* (=) *h2 h1*
 \langle *proof* \rangle

lemma *add-heaps-asso*:
 (*h1 ++ h2*) ++ *h3* = *h1 ++ (h2 ++ h3)*
 \langle *proof* \rangle

lemma *compatible-maps-same*:
 assumes *compatible-maps* (=) *ha hb*
 and *ha x = Some y*
 shows (*ha ++ hb*) *x = Some y*
 \langle *proof* \rangle

lemma *compatible-maps-refl*:
 compatible-maps (=) *h h*
 \langle *proof* \rangle

lemma *map-invo*:
 h ++ h = h
 \langle *proof* \rangle

lemma *included-then-compatible-maps*:
 assumes *map-included h1 h*
 and *map-included h2 h*
 shows *compatible-maps* (=) *h1 h2*
 \langle *proof* \rangle

lemma *commut-charact*:
 assumes *compatible-maps* (=) *h1 h2*
 shows *h1 ++ h2 = h2 ++ h1*
 \langle *proof* \rangle

end

1.2 Fractional Permissions

In this file, we define the type of positive rationals, which we use as permission amounts in extended heaps (see `FractionalHeap.thy`).

```
theory PosRat  
  imports Main HOL.Rat  
begin  
  
  typedef prat = { r :: rat | r. r > 0 }  $\langle$ proof $\rangle$   
  
  setup-lifting type-definition-prat  
  
  lift-definition pwrite :: prat is 1  $\langle$ proof $\rangle$ 
```

lift-definition *half* :: *prat* **is** $1 / 2$ \langle *proof* \rangle

lift-definition *pgte* :: *prat* \Rightarrow *prat* \Rightarrow *bool* **is** (\geq) \langle *proof* \rangle

lift-definition *pgt* :: *prat* \Rightarrow *prat* \Rightarrow *bool* **is** $(>)$ \langle *proof* \rangle

lift-definition *lt* :: *prat* \Rightarrow *prat* \Rightarrow *bool* **is** $(<)$ \langle *proof* \rangle

lift-definition *pmult* :: *prat* \Rightarrow *prat* \Rightarrow *prat* **is** $(*)$ \langle *proof* \rangle

lift-definition *padd* :: *prat* \Rightarrow *prat* \Rightarrow *prat* **is** $(+)$ \langle *proof* \rangle

lift-definition *pdiv* :: *prat* \Rightarrow *prat* \Rightarrow *prat* **is** $(/)$ \langle *proof* \rangle

lift-definition *pmin* :: *prat* \Rightarrow *prat* \Rightarrow *prat* **is** (\min) \langle *proof* \rangle

lift-definition *pmax* :: *prat* \Rightarrow *prat* \Rightarrow *prat* **is** (\max) \langle *proof* \rangle

lemma *pmin-comm*:

pmin *a* *b* = *pmin* *b* *a*
 \langle *proof* \rangle

lemma *pmin-greater*:

pgte *a* (*pmin* *a* *b*)
 \langle *proof* \rangle

lemma *pmin-is*:

assumes *pgte* *a* *b*
shows *pmin* *a* *b* = *b*
 \langle *proof* \rangle

lemma *pmax-comm*:

pmax *a* *b* = *pmax* *b* *a*
 \langle *proof* \rangle

lemma *pmax-smaller*:

pgte (*pmax* *a* *b*) *a*
 \langle *proof* \rangle

lemma *pmax-is*:

assumes *pgte* *a* *b*
shows *pmax* *a* *b* = *a*
 \langle *proof* \rangle

lemma *pmax-is-smaller*:

assumes *pgte* *x* *a*
and *pgte* *x* *b*
shows *pgte* *x* (*pmax* *a* *b*)
 \langle *proof* \rangle

lemma *half-between-0-1*:

pgt pwrite half
<proof>

lemma *pgt-implies-pgte:*
assumes *pgt a b*
shows *pgte a b*
<proof>

lemma *half-plus-half:*
padd half half = pwrite
<proof>

lemma *padd-comm:*
padd a b = padd b a
<proof>

lemma *padd-asso:*
padd (padd a b) c = padd a (padd b c)
<proof>

lemma *pgte-antisym:*
assumes *pgte a b*
and *pgte b a*
shows *a = b*
<proof>

lemma *sum-larger:*
pgt (padd a b) a
<proof>

lemma *greater-sum-both:*
assumes *pgte a (padd b c)*
shows $\exists a1 a2. a = padd a1 a2 \wedge pgte a1 b \wedge pgte a2 c$
<proof>

lemma *padd-cancellative:*
assumes *a = padd x b*
and *a = padd y b*
shows *x = y*
<proof>

lemma *not-pgte-charact:*
 $\neg pgte a b \longleftrightarrow pgt b a$
<proof>

lemma *pgte-pgt:*
assumes *pgt a b*

and $pgte\ c\ d$
shows $pgt\ (padd\ a\ c)\ (padd\ b\ d)$
 $\langle proof \rangle$

lemma $pmult-distr$:
 $pmult\ a\ (padd\ b\ c) = padd\ (pmult\ a\ b)\ (pmult\ a\ c)$
 $\langle proof \rangle$

lemma $pmult-comm$:
 $pmult\ a\ b = pmult\ b\ a$
 $\langle proof \rangle$

lemma $pmult-special$:
 $pmult\ pwrite\ x = x$
 $\langle proof \rangle$

definition $pinv$ **where**
 $pinv\ p = pdiv\ pwrite\ p$

lemma $pinv-double-half$:
 $pmult\ half\ (pinv\ p) = pinv\ (padd\ p\ p)$
 $\langle proof \rangle$

lemma $pinv-inverts$:
assumes $pgte\ a\ b$
shows $pgte\ (pinv\ b)\ (pinv\ a)$
 $\langle proof \rangle$

lemma $pinv-pmult-ok$:
 $pmult\ p\ (pinv\ p) = pwrite$
 $\langle proof \rangle$

lemma $pinv-pwrite$:
 $pinv\ pwrite = pwrite$
 $\langle proof \rangle$

lemma $pmin-pmax$:
assumes $pgte\ x\ (pmin\ a\ b)$
shows $x = pmin\ (pmax\ x\ a)\ (pmax\ x\ b)$
 $\langle proof \rangle$

lemma $pmin-sum$:
 $padd\ (pmin\ a\ b)\ c = pmin\ (padd\ a\ c)\ (padd\ b\ c)$
 $\langle proof \rangle$

lemma *pmin-sum-larger*:
pgte (pmin (padd a1 b1) (padd a2 b2)) (padd (pmin a1 a2) (pmin b1 b2))
<proof>

end

1.3 Permission Heaps

In this file, we define permission heaps, (partial) addition between them, and prove useful lemmas.

theory *FractionalHeap*
imports *Main PosRat PartialMap*
begin

type-synonym *('l, 'v) fract-heap* = *'l* \rightarrow *prat* \times *'v*

Because fractional permissions are at most 1, two permission amounts are compatible if they sum to at most 1.

definition *compatible-fractions* :: *('l, 'v) fract-heap* \Rightarrow *('l, 'v) fract-heap* \Rightarrow *bool*
where

compatible-fractions h h' \longleftrightarrow
($\forall l p p'. h l = \text{Some } p \wedge h' l = \text{Some } p' \longrightarrow \text{pgte } p\text{write } (\text{padd } (\text{fst } p) (\text{fst } p'))$)

definition *same-values* :: *('l, 'v) fract-heap* \Rightarrow *('l, 'v) fract-heap* \Rightarrow *bool* **where**
same-values h h' \longleftrightarrow *($\forall l p p'. h l = \text{Some } p \wedge h' l = \text{Some } p' \longrightarrow \text{snd } p = \text{snd } p'$)*

fun *fadd-options* :: *(prat \times 'v) option* \Rightarrow *(prat \times 'v) option* \Rightarrow *(prat \times 'v) option*
where
fadd-options None x = *x*
 | *fadd-options x None* = *x*
 | *fadd-options (Some x) (Some y)* = *Some (padd (fst x) (fst y), snd x)*

lemma *fadd-options-cancellative*:
assumes *fadd-options a x = fadd-options b x*
shows *a = b*
<proof>

definition *compatible-fract-heaps* :: *('l, 'v) fract-heap* \Rightarrow *('l, 'v) fract-heap* \Rightarrow *bool*
where

compatible-fract-heaps h h' \longleftrightarrow *compatible-fractions h h' \wedge same-values h h'*

lemma *compatible-fract-heapsI*:
assumes $\bigwedge l p p'. h l = \text{Some } p \wedge h' l = \text{Some } p' \implies \text{pgte } p\text{write } (\text{padd } (\text{fst } p) (\text{fst } p'))$
(fst p')

and $\bigwedge l p p'. h l = \text{Some } p \wedge h' l = \text{Some } p' \implies \text{snd } p = \text{snd } p'$
shows *compatible-fract-heaps* $h h'$
 $\langle \text{proof} \rangle$

lemma *compatible-fract-heapsE*:
assumes *compatible-fract-heaps* $h h'$
and $h l = \text{Some } p \wedge h' l = \text{Some } p'$
shows $\text{pgte } p \text{write } (\text{padd } (\text{fst } p) (\text{fst } p'))$
and $\text{snd } p = \text{snd } p'$
 $\langle \text{proof} \rangle$

lemma *compatible-fract-heaps-comm*:
assumes *compatible-fract-heaps* $h h'$
shows *compatible-fract-heaps* $h' h$
 $\langle \text{proof} \rangle$

The following definition of the sum of two permission heaps only makes sense if h and h' are compatible

definition *add-fh* :: $(l, 'v)$ *fract-heap* $\Rightarrow (l, 'v)$ *fract-heap* $\Rightarrow (l, 'v)$ *fract-heap*
where
 $\text{add-fh } h h' l = \text{fadd-options } (h l) (h' l)$

definition *full-ownership* :: $(l, 'v)$ *fract-heap* $\Rightarrow \text{bool}$ **where**
 $\text{full-ownership } h \iff (\forall l p. h l = \text{Some } p \longrightarrow \text{fst } p = \text{pwrite})$

lemma *full-ownershipI*:
assumes $\bigwedge l p. h l = \text{Some } p \implies \text{fst } p = \text{pwrite}$
shows *full-ownership* h
 $\langle \text{proof} \rangle$

fun *apply-opt* **where**
 $\text{apply-opt } f \text{ None} = \text{None}$
 $|\ \text{apply-opt } f (\text{Some } x) = \text{Some } (f x)$

This function maps a permission heap to a normal partial heap (without permissions).

definition *normalize* :: $(l, 'v)$ *fract-heap* $\Rightarrow (l \rightarrow 'v)$ **where**
 $\text{normalize } h l = \text{apply-opt } \text{snd } (h l)$

lemma *normalize-eq*:
 $\text{normalize } h l = \text{None} \iff h l = \text{None}$
 $\text{normalize } h l = \text{Some } v \iff (\exists p. h l = \text{Some } (p, v)) \text{ (is } ?A \iff ?B)$
 $\langle \text{proof} \rangle$

definition *fpdom* **where**
 $\text{fpdom } h = \{x. \exists v. h x = \text{Some } (\text{pwrite}, v)\}$

lemma *compatible-then-dom-disjoint*:

```

assumes compatible-fract-heaps h1 h2
shows dom h1 ∩ fpdom h2 = {}
and dom h2 ∩ fpdom h1 = {}
⟨proof⟩

```

```

lemma compatible-dom-sum:
assumes compatible-fract-heaps h1 h2
shows dom (add-fh h1 h2) = dom h1 ∪ dom h2 (is ?A = ?B)
⟨proof⟩

```

Addition of permission heaps is associative.

```

lemma add-fh-asso:
  add-fh (add-fh a b) c = add-fh a (add-fh b c)
⟨proof⟩

```

```

lemma add-fh-update:
assumes b x = None
shows add-fh (a(x ↦ p)) b = (add-fh a b)(x ↦ p)
⟨proof⟩

```

end

1.4 Extended Heaps

In this file, we define extended heaps, which are triples of a permission heap, a shared action guard state, and a family of unique action guard states. We also define a (partial) addition of two extended heaps. Finally, we prove useful lemmas about them.

```

theory StateModel
imports FractionalHeap HOL-Library.Multiset
begin

```

```

type-synonym loc = nat
type-synonym val = nat

```

We store the initial value with the unique guard

```

type-synonym f-heap = (loc, val) fract-heap
type-synonym 'a gs-heap = (prat × 'a multiset) option
type-synonym ('i, 'a) gu-heap = 'i → 'a list

```

```

type-synonym ('i, 'a) heap = f-heap × 'a gs-heap × ('i, 'a) gu-heap

```

```

type-synonym var = string
type-synonym normal-heap = (nat → nat)
type-synonym store = (var ⇒ nat)

```

```

fun get-fh where get-fh x = fst x

```

```

fun get-gs where get-gs x = fst (snd x)
fun get-gu where get-gu x = snd (snd x)

```

Two "heaps" are compatible iff: 1. The fractional heaps have the same common values and sum to at most 1 2. The unique guard heaps are disjoint 3. The shared guards permissions sum to at most 1

```

definition compatible :: ('i, 'a) heap => ('i, 'a) heap => bool (infixl <##> 60)
where
  h ## h' <=> compatible-fract-heaps (get-fh h) (get-fh h') & (forall k. get-gu h k =
None & get-gu h' k = None)
  & (forall p p'. get-gs h = Some p & get-gs h' = Some p' -> pgte pwrite (padd (fst p)
(fst p')))

```

lemma compatibleI:

```

assumes compatible-fract-heaps (get-fh h) (get-fh h')
and forall k. get-gu h k = None & get-gu h' k = None
and forall p p'. get-gs h = Some p & get-gs h' = Some p' => pgte pwrite (padd
(fst p) (fst p'))
shows h ## h'
<proof>

```

fun add-gu-single **where**

```

  add-gu-single None x = x
| add-gu-single x None = x

```

definition add-gu **where**

```

  add-gu u1 u2 k = add-gu-single (u1 k) (u2 k)

```

lemma comp-add-gu-comm:

```

assumes forall k. h k = None & h' k = None
shows add-gu h h' = add-gu h' h
<proof>

```

fun add-gs :: (prat × 'a multiset) option => (prat × 'a multiset) option => (prat × 'a multiset) option

```

where
  add-gs None x = x
| add-gs x None = x
| add-gs (Some p) (Some p') = Some (padd (fst p) (fst p'), snd p + snd p')

```

Addition of shared guard states is cancellative.

lemma add-gs-cancellative:

```

assumes add-gs a x = add-gs b x
shows a = b
<proof>

```

Addition of shared guard states is commutative.

lemma add-gs-comm:

$add\text{-}gs\ a\ b = add\text{-}gs\ b\ a$
 $\langle proof \rangle$

lemma *compatible-fheaps-comm*:
assumes *compatible-fract-heaps* $a\ b$
shows $add\text{-}fh\ a\ b = add\text{-}fh\ b\ a$
 $\langle proof \rangle$

The following function defines addition between two extended heaps.

fun *plus* :: $('i, 'a)\ heap\ option \Rightarrow ('i, 'a)\ heap\ option \Rightarrow ('i, 'a)\ heap\ option$ (**infixl**
 $\langle \oplus \rangle\ 63$) **where**
 $None \oplus - = None$
 $| - \oplus None = None$
 $| Some\ h1 \oplus Some\ h2 = (if\ h1\ \#\# \ h2\ then\ Some\ (add\text{-}fh\ (get\text{-}fh\ h1)\ (get\text{-}fh\ h2),$
 $add\text{-}gs\ (get\text{-}gs\ h1)\ (get\text{-}gs\ h2),\ add\text{-}gu\ (get\text{-}gu\ h1)\ (get\text{-}gu\ h2))\ else\ None)$

lemma *plus-extract*:
assumes $Some\ x = Some\ a \oplus Some\ b$
shows $get\text{-}fh\ x = add\text{-}fh\ (get\text{-}fh\ a)\ (get\text{-}fh\ b)$
and $get\text{-}gs\ x = add\text{-}gs\ (get\text{-}gs\ a)\ (get\text{-}gs\ b)$
and $get\text{-}gu\ x = add\text{-}gu\ (get\text{-}gu\ a)\ (get\text{-}gu\ b)$
 $\langle proof \rangle$

lemma *compatible-eq*:
 $Some\ a \oplus Some\ b = None \iff \neg a\ \#\# \ b$
 $\langle proof \rangle$

lemma *compatible-comm*:
 $a\ \#\# \ b \iff b\ \#\# \ a$
 $\langle proof \rangle$

lemma *heap-ext*:
assumes $get\text{-}fh\ a = get\text{-}fh\ b$
and $get\text{-}gs\ a = get\text{-}gs\ b$
and $get\text{-}gu\ a = get\text{-}gu\ b$
shows $a = b$
 $\langle proof \rangle$

Addition of two extended heaps is commutative.

lemma *plus-comm*:
 $a \oplus b = b \oplus a$
 $\langle proof \rangle$

lemma *asso2*:
assumes $Some\ a \oplus Some\ b = Some\ ab$
and $\neg b\ \#\# \ c$
shows $\neg ab\ \#\# \ c$
 $\langle proof \rangle$

lemma *plus-extract-point-fh*:

assumes $\text{Some } x = \text{Some } a \oplus \text{Some } b$
and $\text{get-fh } a \ l = \text{Some } pa$
and $\text{get-fh } b \ l = \text{Some } pb$
shows $\text{snd } pa = \text{snd } pb \wedge \text{pgte } pwrite \ (\text{padd } (\text{fst } pa) \ (\text{fst } pb)) \wedge \text{get-fh } x \ l =$
 $\text{Some } (\text{padd } (\text{fst } pa) \ (\text{fst } pb), \text{snd } pa)$
<proof>

lemma *asso1*:

assumes $\text{Some } a \oplus \text{Some } b = \text{Some } ab$
and $\text{Some } b \oplus \text{Some } c = \text{Some } bc$
shows $\text{Some } ab \oplus \text{Some } c = \text{Some } a \oplus \text{Some } bc$
<proof>

lemma *simpler-asso*:

$(\text{Some } a \oplus \text{Some } b) \oplus \text{Some } c = \text{Some } a \oplus (\text{Some } b \oplus \text{Some } c)$
<proof>

Addition of two extended heaps is associative.

lemma *plus-asso*:

$(a \oplus b) \oplus c = a \oplus (b \oplus c)$
<proof>

We define the extension order between extended heaps.

definition *larger* :: $('i, 'a) \text{ heap} \Rightarrow ('i, 'a) \text{ heap} \Rightarrow \text{bool}$ (**infixl** $\langle \succeq \rangle$ 55) **where**
 $a \succeq b \longleftrightarrow (\exists c. \text{Some } a = \text{Some } b \oplus \text{Some } c)$

The extension order between extended heaps is transitive.

lemma *larger-trans*:

assumes $a \succeq b$
and $b \succeq c$
shows $a \succeq c$
<proof>

lemma *comp-smaller*:

assumes $a \#\# b$
and $\text{Some } b = \text{Some } c \oplus \text{Some } d$
shows $a \#\# c$
<proof>

lemma *full-sguard-sum-same*:

assumes $\text{get-gs } a = \text{Some } (pwrite, \text{sargs})$
and $\text{Some } h = \text{Some } a \oplus \text{Some } b$
shows $\text{get-gs } h = \text{Some } (pwrite, \text{sargs})$
<proof>

lemma *full-uguard-sum-same*:

assumes $\text{get-gu } a \ k = \text{Some } uargs$
and $\text{Some } h = \text{Some } a \oplus \text{Some } b$

shows $get\text{-}gu\ h\ k = Some\ uargs$
 $\langle proof \rangle$

lemma *smaller-more-compatible*:

assumes $a \#\# b$
and $a \succeq c$
shows $c \#\# b$
 $\langle proof \rangle$

lemma *equiv-sum-get-fh*:

assumes $get\text{-}fh\ a = get\text{-}fh\ a'$
and $get\text{-}fh\ b = get\text{-}fh\ b'$
and $Some\ x = Some\ a \oplus Some\ b$
and $Some\ x' = Some\ a' \oplus Some\ b'$
shows $get\text{-}fh\ x = get\text{-}fh\ x'$
 $\langle proof \rangle$

lemma *addition-cancellative*:

assumes $Some\ a = Some\ b \oplus Some\ c$
and $Some\ a = Some\ b' \oplus Some\ c$
shows $b = b'$
 $\langle proof \rangle$

lemma *addition-cancellative3*:

assumes $Some\ x = Some\ a \oplus Some\ b \oplus Some\ c$
and $Some\ x = Some\ a' \oplus Some\ b \oplus Some\ c$
shows $a = a'$
 $\langle proof \rangle$

lemma *larger3*:

assumes $Some\ x = Some\ a \oplus Some\ b \oplus Some\ c$
shows $x \succeq b$
 $\langle proof \rangle$

lemma *add-get-fh*:

assumes $Some\ x = Some\ a \oplus Some\ b$
shows $get\text{-}fh\ x = add\text{-}fh\ (get\text{-}fh\ a)\ (get\text{-}fh\ b)$
 $\langle proof \rangle$

lemma *sum-gs-one-none*:

assumes $Some\ x = Some\ a \oplus Some\ b$
and $get\text{-}gs\ b = None$
shows $get\text{-}gs\ x = get\text{-}gs\ a$
 $\langle proof \rangle$

lemma *sum-gs-one-some*:
assumes $\text{Some } x = \text{Some } a \oplus \text{Some } b$
and $\text{get-gs } a = \text{Some } (pa, ma)$
and $\text{get-gs } b = \text{Some } (pb, mb)$
shows $\text{get-gs } x = \text{Some } (\text{padd } pa \ pb, ma + mb)$
 $\langle \text{proof} \rangle$

definition *empty-heap* :: $(\text{'i}, \text{'a}) \text{ heap}$ **where**
 $\text{empty-heap} = (\text{Map.empty}, \text{None}, \lambda k. \text{None})$

lemma *dom-normalize*:
 $\text{dom } h = \text{dom } (\text{normalize } h)$
 $\langle \text{proof} \rangle$

lemma *sum-second-none-get-fh*:
assumes $\text{Some } x = \text{Some } a \oplus \text{Some } b$
and $\text{get-fh } b \ l = \text{None}$
shows $\text{get-fh } x \ l = \text{get-fh } a \ l$
 $\langle \text{proof} \rangle$

lemma *sum-first-none-get-fh*:
assumes $\text{Some } x = \text{Some } a \oplus \text{Some } b$
and $\text{get-fh } a \ l = \text{None}$
shows $\text{get-fh } x \ l = \text{get-fh } b \ l$
 $\langle \text{proof} \rangle$

lemma *dom-sum-two*:
assumes $\text{Some } x = \text{Some } a \oplus \text{Some } b$
shows $\text{dom } (\text{get-fh } x) = \text{dom } (\text{get-fh } a) \cup \text{dom } (\text{get-fh } b)$
 $\langle \text{proof} \rangle$

lemma *dom-three-sum*:
assumes $\text{Some } x = \text{Some } a \oplus \text{Some } b \oplus \text{Some } c$
shows $\text{dom } (\text{get-fh } x) = \text{dom } (\text{get-fh } a) \cup \text{dom } (\text{get-fh } b) \cup \text{dom } (\text{get-fh } c)$
 $\langle \text{proof} \rangle$

lemma *addition-smaller-domain*:
assumes $\text{Some } a = \text{Some } b \oplus \text{Some } c$
shows $\text{dom } (\text{get-fh } b) \subseteq \text{dom } (\text{get-fh } a)$
 $\langle \text{proof} \rangle$

lemma *one-value-sum-same*:
assumes $\text{Some } x = \text{Some } a \oplus \text{Some } b$
and $\text{get-fh } a \ l = \text{Some } (\pi, v)$
shows $\exists \pi'. \text{get-fh } x \ l = \text{Some } (\pi', v)$

<proof>

lemma *compatible-last-two*:

assumes *Some x = Some a \oplus Some b \oplus Some c*

shows *b $\#\#$ c*

<proof>

lemma *add-fh-map-empty*:

add-fh h Map.empty = h

<proof>

definition *bounded where*

bounded h \longleftrightarrow ($\forall l p. \text{fst } h \ l = \text{Some } p \longrightarrow \text{pgte } p \text{write } (\text{fst } p)$)

lemma *boundedI*:

assumes $\bigwedge l p. \text{fst } h \ l = \text{Some } p \implies \text{pgte } p \text{write } (\text{fst } p)$

shows *bounded h*

<proof>

lemma *boundedE*:

assumes *bounded h*

and *fst h l = Some p*

shows *pgte p write (fst p)*

<proof>

lemma *bounded-smaller-sum*:

assumes *bounded x*

and *Some x = Some a \oplus Some b*

shows *bounded a*

<proof>

lemma *bounded-smaller*:

assumes *bounded x*

and $x \succeq a$

shows *bounded a*

<proof>

lemma *sum-perm-smaller*:

assumes *Some x = Some a \oplus Some b*

and *fst a l = Some (p, v)*

shows $\exists p'. \text{pgte } p' \ p \wedge \text{fst } x \ l = \text{Some } (p', v)$

<proof>

lemma *modus-ponens*:

assumes *A*

and $A \implies B$

shows *B*

<proof>

lemma *fpdom-inclusion:*

assumes *Some h' = Some h \oplus Some r*
and *bounded h'*
shows *fpdom (fst h) \subseteq fpdom (fst h')*
<proof>

lemma *fpdom-dom-disjoint:*

assumes *Some h = Some h1 \oplus Some h2*
shows *dom (fst h1) \cap fpdom (fst h2) = {}*
<proof>

lemma *fpdom-dom-union:*

assumes *Some h = Some h1 \oplus Some h2*
and *bounded h*
shows *fpdom (fst h1) \cup fpdom (fst h2) \subseteq fpdom (fst h)*
<proof>

lemma *full-ownership-then-bounded:*

assumes *full-ownership (fst h)*
shows *bounded h*
<proof>

end

2 Imperative Concurrent Language

This file defines the syntax and semantics of the concurrent programming language described in the paper, based on Viktor Vafeiadis' Isabelle soundness proof of CSL [2], and adapted to Isabelle 2016-1 by Qin Yu and James Brotherston (see <https://people.mpi-sws.org/~viktor/csksound/>). We also prove some useful lemmas about the semantics.

theory *Lang*
imports *Main StateModel*
begin

2.1 Language Syntax and Semantics

type-synonym *state = store \times normal-heap*

datatype *exp =*
 Evar var
 | *Enum nat*
 | *Eplus exp exp*

```

datatype bexp =
  | Beq exp exp
  | Band bexp bexp
  | Bnot bexp
  | Btrue

```

```

datatype cmd =
  | Cskip
  | Cassign var exp
  | Cread var exp
  | Cwrite exp exp
  | Calloc var exp
  | Cdispose exp
  | Cseq cmd cmd
  | Cpar cmd cmd
  | Cif bexp cmd cmd
  | Cwhile bexp cmd
  | Catomic cmd

```

Arithmetic expressions (*exp*) consist of variables, constants, and arithmetic operations. Boolean expressions (*bexp*) consist of comparisons between arithmetic expressions. Commands (*cmd*) include the empty command, variable assignments, memory reads, writes, allocations and deallocations, sequential and parallel composition, conditionals, while loops, local variable declarations, and atomic statements.

2.1.1 Semantics of expressions

Denotational semantics for arithmetic and boolean expressions.

primrec

edenot :: *exp* \Rightarrow *store* \Rightarrow *nat*

where

```

  | edenot (Evar v) s      = s v
  | edenot (Enum n) s     = n
  | edenot (Eplus e1 e2) s = edenot e1 s + edenot e2 s

```

primrec

bdenot :: *bexp* \Rightarrow *store* \Rightarrow *bool*

where

```

  | bdenot (Beq e1 e2) s = (edenot e1 s = edenot e2 s)
  | bdenot (Band b1 b2) s = (bdenot b1 s  $\wedge$  bdenot b2 s)
  | bdenot (Bnot b) s    = ( $\neg$  bdenot b s)
  | bdenot Btrue -      = True

```

2.1.2 Semantics of commands

We give a standard small-step operational semantics to commands with configurations being command-state pairs.

inductive

$red :: cmd \Rightarrow state \Rightarrow cmd \Rightarrow state \Rightarrow bool$

and $red\text{-}rtrans :: cmd \Rightarrow state \Rightarrow cmd \Rightarrow state \Rightarrow bool$

where

$red\text{-}Seq1[intro]: red (Cseq Cskip C) \sigma C \sigma$
 $| red\text{-}Seq2[elim]: red C1 \sigma C1' \sigma' \Longrightarrow red (Cseq C1 C2) \sigma (Cseq C1' C2) \sigma'$
 $| red\text{-}If1[intro]: bdenot B (fst \sigma) \Longrightarrow red (Cif B C1 C2) \sigma C1 \sigma$
 $| red\text{-}If2[intro]: \neg bdenot B (fst \sigma) \Longrightarrow red (Cif B C1 C2) \sigma C2 \sigma$
 $| red\text{-}Atomic[intro]: red\text{-}rtrans C \sigma Cskip \sigma' \Longrightarrow red (Catomic C) \sigma Cskip \sigma'$
 $| red\text{-}Par1[elim]: red C1 \sigma C1' \sigma' \Longrightarrow red (Cpar C1 C2) \sigma (Cpar C1' C2) \sigma'$
 $| red\text{-}Par2[elim]: red C2 \sigma C2' \sigma' \Longrightarrow red (Cpar C1 C2) \sigma (Cpar C1 C2') \sigma'$
 $| red\text{-}Par3[intro]: red (Cpar Cskip Cskip) \sigma (Cskip) \sigma$
 $| red\text{-}Loop[intro]: red (Cwhile B C) \sigma (Cif B (Cseq C (Cwhile B C)) Cskip) \sigma$
 $| red\text{-}Assign[intro]: \llbracket \sigma = (s,h); \sigma' = (s(x := edenot E s), h) \rrbracket \Longrightarrow red (Cassign x E) \sigma Cskip \sigma'$
 $| red\text{-}Read[intro]: \llbracket \sigma = (s,h); h(edenot E s) = Some v; \sigma' = (s(x := v), h) \rrbracket \Longrightarrow red (Cread x E) \sigma Cskip \sigma'$
 $| red\text{-}Write[intro]: \llbracket \sigma = (s,h); \sigma' = (s, h(edenot E s \mapsto edenot E' s)) \rrbracket \Longrightarrow red (Cwrite E E') \sigma Cskip \sigma'$
 $| red\text{-}Alloc[intro]: \llbracket \sigma = (s,h); v \notin dom h; \sigma' = (s(x := v), h(v \mapsto edenot E s)) \rrbracket \Longrightarrow red (Calloc x E) \sigma Cskip \sigma'$
 $| red\text{-}Free[intro]: \llbracket \sigma = (s,h); \sigma' = (s, h(edenot E s := None)) \rrbracket \Longrightarrow red (Cdispose E) \sigma Cskip \sigma'$

 $| NoStep: red\text{-}rtrans C \sigma C \sigma$
 $| OneMoreStep: \llbracket red C \sigma C' \sigma'; red\text{-}rtrans C' \sigma' C'' \sigma'' \rrbracket \Longrightarrow red\text{-}rtrans C \sigma C'' \sigma''$

inductive-cases $red\text{-}par\text{-}cases: red (Cpar C1 C2) \sigma C' \sigma'$

inductive-cases $red\text{-}atomic\text{-}cases: red (Catomic C) \sigma C' \sigma'$

2.1.3 Abort semantics

primrec

$accesses :: cmd \Rightarrow store \Rightarrow nat set$

where

$accesses Cskip \quad s = \{\}$
 $| accesses (Cassign x E) \quad s = \{\}$
 $| accesses (Cread x E) \quad s = \{edenot E s\}$
 $| accesses (Cwrite E E') \quad s = \{edenot E s\}$
 $| accesses (Calloc x E) \quad s = \{\}$
 $| accesses (Cdispose E) \quad s = \{edenot E s\}$
 $| accesses (Cseq C1 C2) \quad s = accesses C1 s$
 $| accesses (Cpar C1 C2) \quad s = accesses C1 s \cup accesses C2 s$
 $| accesses (Cif B C1 C2) \quad s = \{\}$
 $| accesses (Cwhile B C) \quad s = \{\}$
 $| accesses (Catomic C) \quad s = \{\}$

primrec

$$\text{writes} :: \text{cmd} \Rightarrow \text{store} \Rightarrow \text{nat set}$$
where

$$\begin{array}{l} \text{writes } C\text{skip} \quad s = \{\} \\ | \text{writes } (C\text{assign } x \ E) \quad s = \{\} \\ | \text{writes } (C\text{read } x \ E) \quad s = \{\} \\ | \text{writes } (C\text{write } E \ E') \quad s = \{\text{edenot } E \ s\} \\ | \text{writes } (C\text{alloc } x \ E) \quad s = \{\} \\ | \text{writes } (C\text{dispose } E) \quad s = \{\text{edenot } E \ s\} \\ | \text{writes } (C\text{seq } C1 \ C2) \quad s = \text{writes } C1 \ s \\ | \text{writes } (C\text{par } C1 \ C2) \quad s = \text{writes } C1 \ s \cup \text{writes } C2 \ s \\ | \text{writes } (C\text{if } B \ C1 \ C2) \quad s = \{\} \\ | \text{writes } (C\text{while } B \ C) \quad s = \{\} \\ | \text{writes } (C\text{atomic } C) \quad s = \{\} \end{array}$$
inductive

$$\text{aborts} :: \text{cmd} \Rightarrow \text{state} \Rightarrow \text{bool}$$
where

$$\begin{array}{l} \text{aborts-Seq[intro]: } \text{aborts } C1 \ \sigma \Longrightarrow \text{aborts } (C\text{seq } C1 \ C2) \ \sigma \\ | \text{aborts-Atomic[intro]: } \llbracket \text{red-rtrans } C \ \sigma \ C' \ \sigma' ; \text{aborts } C' \ \sigma' \rrbracket \Longrightarrow \text{aborts } (C\text{atomic } \\ C) \ \sigma \\ | \text{aborts-Par1[intro]: } \text{aborts } C1 \ \sigma \Longrightarrow \text{aborts } (C\text{par } C1 \ C2) \ \sigma \\ | \text{aborts-Par2[intro]: } \text{aborts } C2 \ \sigma \Longrightarrow \text{aborts } (C\text{par } C1 \ C2) \ \sigma \\ | \text{aborts-Read[intro]: } \text{edenot } E \ (\text{fst } \sigma) \notin \text{dom } (\text{snd } \sigma) \Longrightarrow \text{aborts } (C\text{read } x \ E) \ \sigma \\ | \text{aborts-Write[intro]: } \text{edenot } E \ (\text{fst } \sigma) \notin \text{dom } (\text{snd } \sigma) \Longrightarrow \text{aborts } (C\text{write } E \ E') \ \sigma \\ | \text{aborts-Free[intro]: } \text{edenot } E \ (\text{fst } \sigma) \notin \text{dom } (\text{snd } \sigma) \Longrightarrow \text{aborts } (C\text{dispose } E) \ \sigma \\ | \text{aborts-Race1[intro]: } \text{accesses } C1 \ (\text{fst } \sigma) \cap \text{writes } C2 \ (\text{fst } \sigma) \neq \{\} \Longrightarrow \text{aborts} \\ (C\text{par } C1 \ C2) \ \sigma \\ | \text{aborts-Race2[intro]: } \text{writes } C1 \ (\text{fst } \sigma) \cap \text{accesses } C2 \ (\text{fst } \sigma) \neq \{\} \Longrightarrow \text{aborts} \\ (C\text{par } C1 \ C2) \ \sigma \end{array}$$

inductive-cases *abort-atomic-cases*: $\text{aborts } (C\text{atomic } C) \ \sigma$

2.2 Useful Definitions and Results

The free variables of expressions, boolean expressions, and commands are defined as expected:

primrec

$$\text{fvE} :: \text{exp} \Rightarrow \text{var set}$$
where

$$\begin{array}{l} \text{fvE } (E\text{var } v) = \{v\} \\ | \text{fvE } (E\text{enum } n) = \{\} \\ | \text{fvE } (E\text{plus } e1 \ e2) = (\text{fvE } e1 \cup \text{fvE } e2) \end{array}$$
primrec

$$\text{fvB} :: \text{bexp} \Rightarrow \text{var set}$$

where

$$\begin{aligned} | \text{fvB } (\text{Beq } e1 \ e2) &= (\text{fvE } e1 \cup \text{fvE } e2) \\ | \text{fvB } (\text{Band } b1 \ b2) &= (\text{fvB } b1 \cup \text{fvB } b2) \\ | \text{fvB } (\text{Bnot } b) &= (\text{fvB } b) \\ | \text{fvB } \text{Btrue} &= \{\} \end{aligned}$$

primrec

$$\text{fvC} :: \text{cmd} \Rightarrow \text{var set}$$

where

$$\begin{aligned} | \text{fvC } (\text{Cskip}) &= \{\} \\ | \text{fvC } (\text{Cassign } v \ E) &= (\{v\} \cup \text{fvE } E) \\ | \text{fvC } (\text{Cread } v \ E) &= (\{v\} \cup \text{fvE } E) \\ | \text{fvC } (\text{Cwrite } E1 \ E2) &= (\text{fvE } E1 \cup \text{fvE } E2) \\ | \text{fvC } (\text{Calloc } v \ E) &= (\{v\} \cup \text{fvE } E) \\ | \text{fvC } (\text{Cdispose } E) &= (\text{fvE } E) \\ | \text{fvC } (\text{Cseq } C1 \ C2) &= (\text{fvC } C1 \cup \text{fvC } C2) \\ | \text{fvC } (\text{Cpar } C1 \ C2) &= (\text{fvC } C1 \cup \text{fvC } C2) \\ | \text{fvC } (\text{Cif } B \ C1 \ C2) &= (\text{fvB } B \cup \text{fvC } C1 \cup \text{fvC } C2) \\ | \text{fvC } (\text{Cwhile } B \ C) &= (\text{fvB } B \cup \text{fvC } C) \\ | \text{fvC } (\text{Catomic } C) &= (\text{fvC } C) \end{aligned}$$

primrec

$$\text{wrC} :: \text{cmd} \Rightarrow \text{var set}$$

where

$$\begin{aligned} | \text{wrC } (\text{Cskip}) &= \{\} \\ | \text{wrC } (\text{Cassign } v \ E) &= \{v\} \\ | \text{wrC } (\text{Cread } v \ E) &= \{v\} \\ | \text{wrC } (\text{Cwrite } E1 \ E2) &= \{\} \\ | \text{wrC } (\text{Calloc } v \ E) &= \{v\} \\ | \text{wrC } (\text{Cdispose } E) &= \{\} \\ | \text{wrC } (\text{Cseq } C1 \ C2) &= (\text{wrC } C1 \cup \text{wrC } C2) \\ | \text{wrC } (\text{Cpar } C1 \ C2) &= (\text{wrC } C1 \cup \text{wrC } C2) \\ | \text{wrC } (\text{Cif } B \ C1 \ C2) &= (\text{wrC } C1 \cup \text{wrC } C2) \\ | \text{wrC } (\text{Cwhile } B \ C) &= (\text{wrC } C) \\ | \text{wrC } (\text{Catomic } C) &= (\text{wrC } C) \end{aligned}$$

primrec

$$\text{subE} :: \text{var} \Rightarrow \text{exp} \Rightarrow \text{exp} \Rightarrow \text{exp}$$

where

$$\begin{aligned} | \text{subE } x \ E \ (\text{Evar } y) &= (\text{if } x = y \ \text{then } E \ \text{else } \text{Evar } y) \\ | \text{subE } x \ E \ (\text{Enum } n) &= \text{Enum } n \\ | \text{subE } x \ E \ (\text{Eplus } e1 \ e2) &= \text{Eplus } (\text{subE } x \ E \ e1) \ (\text{subE } x \ E \ e2) \end{aligned}$$

primrec

$$\text{subB} :: \text{var} \Rightarrow \text{exp} \Rightarrow \text{bexp} \Rightarrow \text{bexp}$$

where

$$\begin{aligned} | \text{subB } x \ E \ (\text{Beq } e1 \ e2) &= \text{Beq } (\text{subE } x \ E \ e1) \ (\text{subE } x \ E \ e2) \\ | \text{subB } x \ E \ (\text{Band } b1 \ b2) &= \text{Band } (\text{subB } x \ E \ b1) \ (\text{subB } x \ E \ b2) \end{aligned}$$

| $subB\ x\ E\ (Bnot\ b) = Bnot\ (subB\ x\ E\ b)$
| $subB\ x\ E\ Btrue = Btrue$

Basic properties of substitutions:

lemma *subE-assign*:

$edenot\ (subE\ x\ E\ e)\ s = edenot\ e\ (s(x := edenot\ E\ s))$
 $\langle proof \rangle$

lemma *subB-assign*:

$bdenot\ (subB\ x\ E\ b)\ s = bdenot\ b\ (s(x := edenot\ E\ s))$
 $\langle proof \rangle$

inductive-cases *red-skip-cases*: $red\ Cskip\ \sigma\ C'\ \sigma'$

inductive-cases *aborts-skip-cases*: $aborts\ Cskip\ \sigma$

lemma *skip-simps[simp]*:

$\neg\ red\ Cskip\ \sigma\ C'\ \sigma'$
 $\neg\ aborts\ Cskip\ \sigma$
 $\langle proof \rangle$

definition

$agrees :: 'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$

where

$agrees\ X\ s\ s' \equiv \forall x \in X. s\ x = s'\ x$

lemma *agrees-union*:

$agrees\ (A \cup B)\ s\ s' \longleftrightarrow agrees\ A\ s\ s' \wedge agrees\ B\ s\ s'$
 $\langle proof \rangle$

Proposition 4.1: Properties of basic properties of *red*.

lemma *agreesI*:

assumes $\bigwedge x. x \in X \implies s\ x = s'\ x$
shows $agrees\ X\ s\ s'$
 $\langle proof \rangle$

lemma *red-properties*:

$red\ C\ \sigma\ C'\ \sigma' \implies fvC\ C' \subseteq fvC\ C \wedge wrC\ C' \subseteq wrC\ C \wedge agrees\ (-\ wrC\ C)\ (fst\ \sigma')\ (fst\ \sigma)$
 $red\ rtrans\ C\ \sigma\ C'\ \sigma' \implies fvC\ C' \subseteq fvC\ C \wedge wrC\ C' \subseteq wrC\ C \wedge agrees\ (-\ wrC\ C)\ (fst\ \sigma')\ (fst\ \sigma)$
 $\langle proof \rangle$

Proposition 4.2: Semantics does not depend on variables not free in the term

lemma *exp-agrees*: $agrees\ (fvE\ E)\ s\ s' \implies edenot\ E\ s = edenot\ E\ s'$

$\langle proof \rangle$

lemma *bexp-agrees*:

$agrees (fvB B) s s' \implies bdenot B s = bdenot B s'$
 ⟨proof⟩

lemma *red-not-in-fv-not-touched*:

$red C \sigma C' \sigma' \implies x \notin fvC C \implies fst \sigma x = fst \sigma' x$
 $red-rtrans C \sigma C' \sigma' \implies x \notin fvC C \implies fst \sigma x = fst \sigma' x$
 ⟨proof⟩

lemma *agrees-update1*:

assumes $agrees X s s'$
shows $agrees X (s(x := v)) (s'(x := v))$
 ⟨proof⟩

lemma *agrees-update2*:

assumes $agrees X s s'$
and $x \notin X$
shows $agrees X (s(x := v)) (s'(x := v'))$
 ⟨proof⟩

lemma *red-agrees-aux*:

$red C \sigma C' \sigma' \implies (\forall s h. agrees X (fst \sigma) s \wedge snd \sigma = h \wedge fvC C \subseteq X \longrightarrow$
 $(\exists s' h'. red C (s, h) C' (s', h') \wedge agrees X (fst \sigma') s' \wedge snd \sigma' = h'))$
 $red-rtrans C \sigma C' \sigma' \implies (\forall s h. agrees X (fst \sigma) s \wedge snd \sigma = h \wedge fvC C \subseteq X$
 \longrightarrow
 $(\exists s' h'. red-rtrans C (s, h) C' (s', h') \wedge agrees X (fst \sigma') s' \wedge snd \sigma' = h'))$
 ⟨proof⟩

lemma *red-agrees[rule-format]*:

$red C \sigma C' \sigma' \implies \forall X s. agrees X (fst \sigma) s \longrightarrow snd \sigma = h \longrightarrow fvC C \subseteq X \longrightarrow$
 $(\exists s' h'. red C (s, h) C' (s', h') \wedge agrees X (fst \sigma') s' \wedge snd \sigma' = h')$
 ⟨proof⟩

lemma *writes-accesses*: $writes C s \subseteq accesses C s$
 ⟨proof⟩

lemma *accesses-agrees*: $agrees (fvC C) s s' \implies accesses C s = accesses C s'$
 ⟨proof⟩

lemma *writes-agrees*: $agrees (fvC C) s s' \implies writes C s = writes C s'$
 ⟨proof⟩

lemma *aborts-agrees*:

assumes $aborts C \sigma$
and $agrees (fvC C) (fst \sigma) s$
and $snd \sigma = h$
shows $aborts C (s, h)$
 ⟨proof⟩

corollary *exp-agrees2[simp]*:
 $x \notin \text{fv}E \ E \implies \text{edenot } E \ (s(x := v)) = \text{edenot } E \ s$
 ⟨proof⟩

lemma *agrees-update*:
assumes $a \notin S$
shows $\text{agrees } S \ s \ (s(a := v))$
 ⟨proof⟩

lemma *agrees-comm*:
 $\text{agrees } S \ s \ s' \longleftrightarrow \text{agrees } S \ s' \ s$
 ⟨proof⟩

lemma *not-in-dom*:
assumes $x \notin \text{dom } s$
shows $s \ x = \text{None}$
 ⟨proof⟩

lemma *agrees-minusD*:
 $\text{agrees } (-X) \ x \ y \implies X \cap Y = \{\} \implies \text{agrees } Y \ x \ y$
 ⟨proof⟩

end

3 CommCSL

In this file, we define the assertion language and the rules of CommCSL.

theory *CommCSL*
imports *Lang StateModel*
begin

definition *no-guard* :: $('i, 'a) \text{ heap} \Rightarrow \text{bool}$ **where**
 $\text{no-guard } h \longleftrightarrow \text{get-gs } h = \text{None} \wedge (\forall k. \text{get-gu } h \ k = \text{None})$

typedef $'a \text{ precondition} = \{ \text{pre} :: ('a \Rightarrow 'a \Rightarrow \text{bool}) \mid \text{pre}. \forall a \ b. \text{pre } a \ b \longrightarrow (\text{pre } b \ a \wedge \text{pre } a \ a) \}$
 ⟨proof⟩

lemma *charact-rep-prec*:
assumes *Rep-precondition* $\text{pre } a \ b$
shows $\text{Rep-precondition } \text{pre } b \ a \wedge \text{Rep-precondition } \text{pre } a \ a$
 ⟨proof⟩

typedef $('i, 'a) \text{ indexed-precondition} = \{ \text{pre} :: ('i \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}) \mid \text{pre}. \forall a \ b \ k. \text{pre } k \ a \ b \longrightarrow (\text{pre } k \ b \ a \wedge \text{pre } k \ a \ a) \}$
 ⟨proof⟩

lemma *charact-rep-indexed-prec*:

assumes *Rep-indexed-precondition pre k a b*

shows *Rep-indexed-precondition pre k b a \wedge Rep-indexed-precondition pre k a a*

<proof>

type-synonym *'a list-exp = store \Rightarrow 'a list*

3.1 Assertion Language

datatype *('i, 'a, 'v) assertion =*

Bool bexp

| *Emp*

| *And ('i, 'a, 'v) assertion ('i, 'a, 'v) assertion*

| *Star ('i, 'a, 'v) assertion ('i, 'a, 'v) assertion (\leftarrow * \rightarrow 70)*

| *Low bexp*

| *LowExp exp*

| *PointsTo exp prat exp*

| *Exists var ('i, 'a, 'v) assertion*

| *EmptyFullGuards*

| *PreSharedGuards 'a precondition*

| *PreUniqueGuards ('i, 'a) indexed-precondition*

| *View normal-heap \Rightarrow 'v ('i, 'a, 'v) assertion store \Rightarrow 'v*

| *SharedGuard prat store \Rightarrow 'a multiset*

| *UniqueGuard 'i 'a list-exp*

| *Imp bexp ('i, 'a, 'v) assertion*

| *NoGuard*

inductive *PRE-shared-simpler :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a multiset \Rightarrow 'a multiset \Rightarrow bool* **where**

Empty: PRE-shared-simpler spre {#} {#}

| *Step: \llbracket PRE-shared-simpler spre a b ; spre xa xb $\rrbracket \Longrightarrow$ PRE-shared-simpler spre (a + {# xa #}) (b + {# xb #})*

definition *PRE-unique :: ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow 'b list \Rightarrow 'b list \Rightarrow bool* **where**

PRE-unique upre uargs uargs' \longleftrightarrow length uargs = length uargs' \wedge ($\forall i. i \geq 0 \wedge i < \text{length } uargs' \longrightarrow \text{upre } (uargs ! i) (uargs' ! i)$)

The following function defines the validity of CommCSL assertions, which corresponds to Figure 7 from the paper.

fun *hyper-sat :: (store \times ('i, 'a) heap) \Rightarrow (store \times ('i, 'a) heap) \Rightarrow ('i, 'a, nat) assertion \Rightarrow bool (\leftarrow , $- \models \rightarrow$ [51, 65, 66] 50)* **where**

$$\begin{aligned}
& |(s, -), (s', -)| \models \text{Bool } b \iff \text{bdenot } b \ s \wedge \text{bdenot } b \ s' \\
& |(-, h), (-, h')| \models \text{Emp} \iff \text{dom } (\text{get-fh } h) = \{\} \wedge \text{dom } (\text{get-fh } h') = \{\} \\
& |\sigma, \sigma'| \models \text{And } A \ B \iff \sigma, \sigma' \models A \wedge \sigma, \sigma' \models B
\end{aligned}$$

$$\begin{aligned}
& |(s, h), (s', h')| \models \text{Star } A \ B \iff (\exists h1 \ h2 \ h1' \ h2'. \text{Some } h = \text{Some } h1 \oplus \text{Some } h2 \wedge \text{Some } h' = \text{Some } h1' \oplus \text{Some } h2') \\
& \quad \wedge |(s, h1), (s', h1')| \models A \wedge |(s, h2), (s', h2')| \models B \\
& |(s, h), (s', h')| \models \text{Low } e \iff \text{bdenot } e \ s = \text{bdenot } e \ s'
\end{aligned}$$

$$\begin{aligned}
& |(s, h), (s', h')| \models \text{PointsTo } \text{loc } p \ x \iff \text{get-fh } h \ (\text{edenot } \text{loc } s) = \text{Some } (p, \text{edenot } x \ s) \wedge \text{get-fh } h' \ (\text{edenot } \text{loc } s') = \text{Some } (p, \text{edenot } x \ s') \\
& \quad \wedge \text{dom } (\text{get-fh } h) = \{\text{edenot } \text{loc } s\} \wedge \text{dom } (\text{get-fh } h') = \{\text{edenot } \text{loc } s'\} \\
& |(s, h), (s', h')| \models \text{Exists } x \ A \iff (\exists v \ v'. (s(x := v), h), (s'(x := v'), h') \models A)
\end{aligned}$$

$$|(s, h), (s', h')| \models \text{EmptyFullGuards} \iff (\text{get-gs } h = \text{Some } (\text{pwrite}, \{\#\}) \wedge (\forall k. \text{get-gu } h \ k = \text{Some } \square)) \wedge (\text{get-gs } h' = \text{Some } (\text{pwrite}, \{\#\}) \wedge (\forall k. \text{get-gu } h' \ k = \text{Some } \square))$$

$$\begin{aligned}
& |(s, h), (s', h')| \models \text{PreSharedGuards } \text{spre} \iff \\
& \quad (\exists \text{sargs } \text{sargs}'. \text{get-gs } h = \text{Some } (\text{pwrite}, \text{sargs}) \wedge \text{get-gs } h' = \text{Some } (\text{pwrite}, \text{sargs}') \wedge \text{PRE-shared-simpler } (\text{Rep-precondition } \text{spre}) \ \text{sargs } \text{sargs}' \\
& \quad \wedge \text{get-fh } h = \text{Map.empty} \wedge \text{get-fh } h' = \text{Map.empty}) \\
& |(s, h), (s', h')| \models \text{PreUniqueGuards } \text{upre} \iff \\
& \quad (\exists \text{uargs } \text{uargs}'. (\forall k. \text{get-gu } h \ k = \text{Some } (\text{uargs } k)) \wedge (\forall k. \text{get-gu } h' \ k = \text{Some } (\text{uargs}' k)) \wedge (\forall k. \text{PRE-unique } (\text{Rep-indexed-precondition } \text{upre } k) \ (\text{uargs } k) \ (\text{uargs}' k)) \wedge \text{get-fh } h = \text{Map.empty} \wedge \text{get-fh } h' = \text{Map.empty})
\end{aligned}$$

$$\begin{aligned}
& |(s, h), (s', h')| \models \text{View } f \ J \ e \iff ((s, h), (s', h') \models J \wedge f \ (\text{normalize } (\text{get-fh } h)) = e \ s \wedge f \ (\text{normalize } (\text{get-fh } h')) = e \ s') \\
& |(s, h), (s', h')| \models \text{SharedGuard } \pi \ ms \iff ((\forall k. \text{get-gu } h \ k = \text{None} \wedge \text{get-gu } h' \ k = \text{None}) \wedge \text{get-gs } h = \text{Some } (\pi, ms \ s) \wedge \text{get-gs } h' = \text{Some } (\pi, ms \ s')) \\
& \quad \wedge \text{get-fh } h = \text{Map.empty} \wedge \text{get-fh } h' = \text{Map.empty})
\end{aligned}$$

$$\begin{aligned}
& |(s, h), (s', h')| \models \text{UniqueGuard } k \ \text{lexp} \iff (\text{get-gs } h = \text{None} \wedge \text{get-gu } h \ k = \text{Some } (\text{lexp } s) \wedge \text{get-gu } h' \ k = \text{Some } (\text{lexp } s')) \wedge \text{get-gs } h' = \text{None} \\
& \quad \wedge \text{get-fh } h = \text{Map.empty} \wedge \text{get-fh } h' = \text{Map.empty} \wedge (\forall k'. k' \neq k \longrightarrow \text{get-gu } h \ k' = \text{None} \wedge \text{get-gu } h' \ k' = \text{None})
\end{aligned}$$

$$|(s, h), (s', h')| \models \text{LowExp } e \iff \text{edenot } e \ s = \text{edenot } e \ s'$$

$$|(s, h), (s', h')| \models \text{Imp } b \ A \iff \text{bdenot } b \ s = \text{bdenot } b \ s' \wedge (\text{bdenot } b \ s \longrightarrow (s, h), (s', h') \models A)$$

$$|(s, h), (s', h')| \models \text{NoGuard} \iff (\text{get-gs } h = \text{None} \wedge (\forall k. \text{get-gu } h \ k = \text{None})) \wedge (\text{get-gs } h' = \text{None} \wedge (\forall k. \text{get-gu } h' \ k = \text{None}))$$

lemma *sat-PreUniqueE*:

assumes $(s, h), (s', h') \models \text{PreUniqueGuards } \text{upre}$

shows $\exists \text{uargs } \text{uargs}'. (\forall k. \text{get-gu } h \ k = \text{Some } (\text{uargs } k)) \wedge (\forall k. \text{get-gu } h' \ k =$

Some (*uargs* 'k) \wedge ($\forall k$. *PRE-unique* (*Rep-indexed-precondition* *upre* k) (*uargs* k)
(*uargs* 'k)
⟨*proof*⟩

lemma *decompose-heap-triple*:
h = (*get-fh* *h*, *get-gs* *h*, *get-gu* *h*)
⟨*proof*⟩

definition *depends-only-on* :: (*store* \Rightarrow 'v) \Rightarrow *var set* \Rightarrow *bool* **where**
depends-only-on *e S* \longleftrightarrow ($\forall s s'$. *agrees* *S s s'* \longrightarrow *e s* = *e s'*)

lemma *depends-only-onI*:
assumes $\bigwedge s s' :: \textit{store. agrees } S s s' \Longrightarrow e s = e s'$
shows *depends-only-on* *e S*
⟨*proof*⟩

definition *fvS* :: (*store* \Rightarrow 'v) \Rightarrow *var set* **where**
fvS *e* = (*SOME* *S. depends-only-on* *e S*)

lemma *fvSE*:
assumes *agrees* (*fvS* *e*) *s s'*
shows *e s* = *e s'*
⟨*proof*⟩

fun *fvA* :: ('i, 'a, 'v) *assertion* \Rightarrow *var set* **where**
fvA (*Bool* *b*) = *fvB* *b*
| *fvA* (*And* *A B*) = *fvA* *A* \cup *fvA* *B*
| *fvA* (*Star* *A B*) = *fvA* *A* \cup *fvA* *B*
| *fvA* (*Low* *e*) = *fvB* *e*
| *fvA* *Emp* = {}
| *fvA* (*PointsTo* *v va vb*) = *fvE* *v* \cup *fvE* *vb*
| *fvA* (*Exists* *x A*) = *fvA* *A* - {*x*}
| *fvA* (*SharedGuard* - *e*) = *fvS* *e*
| *fvA* (*UniqueGuard* - *e*) = *fvS* *e*
| *fvA* (*View* *view A e*) = *fvA* *A* \cup *fvS* *e*
| *fvA* (*PreSharedGuards* -) = {}
| *fvA* (*PreUniqueGuards* -) = {}
| *fvA* *EmptyFullGuards* = {}
| *fvA* (*LowExp* *x*) = *fvE* *x*
| *fvA* (*Imp* *b A*) = *fvB* *b* \cup *fvA* *A*

definition *subS* :: *var* \Rightarrow *exp* \Rightarrow (*store* \Rightarrow 'v) \Rightarrow (*store* \Rightarrow 'v) **where**
subS *x E e* = (λs . *e* (*s*(*x* := *edenot* *E s*)))

lemma *subS-assign*:
subS *x E e s* \longleftrightarrow *e* (*s*(*x* := *edenot* *E s*))

<proof>

fun *collect-existentials* :: ('i, 'a, nat) assertion \Rightarrow var set **where**
 collect-existentials (And A B) = *collect-existentials* A \cup *collect-existentials* B
| *collect-existentials* (Star A B) = *collect-existentials* A \cup *collect-existentials* B
| *collect-existentials* (Exists x A) = *collect-existentials* A \cup {x}
| *collect-existentials* (View view A e) = *collect-existentials* A
| *collect-existentials* (Imp - A) = *collect-existentials* A
| *collect-existentials* - = {}

fun *subA* :: var \Rightarrow exp \Rightarrow ('i, 'a, nat) assertion \Rightarrow ('i, 'a, nat) assertion **where**
 subA x E (And A B) = And (*subA* x E A) (*subA* x E B)
| *subA* x E (Star A B) = Star (*subA* x E A) (*subA* x E B)
| *subA* x E (Bool B) = Bool (*subB* x E B)
| *subA* x E (Low e) = Low (*subB* x E e)
| *subA* x E (LowExp e) = LowExp (*subE* x E e)
| *subA* x E (UniqueGuard k e) = UniqueGuard k (*subS* x E e)
| *subA* x E (SharedGuard π e) = SharedGuard π (*subS* x E e)
| *subA* x E (View view A e) = View view (*subA* x E A) (*subS* x E e)
| *subA* x E (PointsTo loc π e) = PointsTo (*subE* x E loc) π (*subE* x E e)
| *subA* x E (Exists y A) = (if x = y then Exists y A else Exists y (*subA* x E A))
| *subA* x E (Imp b A) = Imp (*subB* x E b) (*subA* x E A)
| *subA* - - A = A

lemma *subA-assign*:

assumes *collect-existentials* A \cap fvE E = {}
 shows (s, h), (s', h') \models *subA* x E A \longleftrightarrow (s(x := edenot E s), h), (s'(x := edenot E s'), h') \models A
 <proof>

lemma *PRE-uniqueI*:

assumes length uargs = length uargs'
 and $\bigwedge i. i \geq 0 \wedge i < \text{length } uargs' \implies \text{upre } (uargs ! i) (uargs' ! i)$
 shows *PRE-unique* upre uargs uargs'
 <proof>

lemma *PRE-unique-implies-tl*:

assumes *PRE-unique* upre (ta # qa) (tb # qb)
 shows *PRE-unique* upre qa qb
 <proof>

lemma *charact-PRE-unique*:

assumes *PRE-unique* (Rep-indexed-precondition pre k) a b
 shows *PRE-unique* (Rep-indexed-precondition pre k) b a \wedge *PRE-unique* (Rep-indexed-precondition pre k) a a
 <proof>

lemma *charact-PRE-shared-simpler*:

assumes *PRE-shared-simpler* $rpre\ a\ b$
and *Rep-precondition* $pre = rpre$
shows *PRE-shared-simpler* (*Rep-precondition* pre) $b\ a \wedge PRE\text{-shared-simpler}$
(*Rep-precondition* pre) $a\ a$
 $\langle proof \rangle$

lemma *always-sat-refl-aux*:
assumes $(s, h), (s', h') \models A$
shows $(s, h), (s, h) \models A$
 $\langle proof \rangle$

lemma *always-sat-refl*:
assumes $\sigma, \sigma' \models A$
shows $\sigma, \sigma \models A$
 $\langle proof \rangle$

lemma *agrees-same-aux*:
assumes *agrees* $(fvA\ A)\ s\ s''$
and $(s, h), (s', h') \models A$
shows $(s'', h), (s', h') \models A$
 $\langle proof \rangle$

lemma *agrees-same*:
assumes *agrees* $(fvA\ A)\ s\ s''$
shows $(s, h), (s', h') \models A \longleftrightarrow (s'', h), (s', h') \models A$
 $\langle proof \rangle$

lemma *sat-comm-aux*:
 $(s, h), (s', h') \models A \implies (s', h'), (s, h) \models A$
 $\langle proof \rangle$

lemma *sat-comm*:
 $\sigma, \sigma' \models A \longleftrightarrow \sigma', \sigma \models A$
 $\langle proof \rangle$

definition *precise where*
 $precise\ J \longleftrightarrow (\forall s1\ H1\ h1\ h1'\ s2\ H2\ h2\ h2'. H1 \succeq h1 \wedge H1 \succeq h1' \wedge H2 \succeq h2$
 $\wedge H2 \succeq h2'$
 $\wedge (s1, h1), (s2, h2) \models J \wedge (s1, h1'), (s2, h2') \models J \implies h1' = h1 \wedge h2' = h2)$

lemma *preciseI*:
assumes $\bigwedge s1\ H1\ h1\ h1'\ s2\ H2\ h2\ h2'. H1 \succeq h1 \wedge H1 \succeq h1' \wedge H2 \succeq h2 \wedge H2$
 $\succeq h2' \implies$
 $(s1, h1), (s2, h2) \models J \implies (s1, h1'), (s2, h2') \models J \implies h1' = h1 \wedge h2' =$
 $h2$
shows *precise* J
 $\langle proof \rangle$

lemma *preciseE*:

assumes *precise J*

and $H1 \succeq h1 \wedge H1 \succeq h1' \wedge H2 \succeq h2 \wedge H2 \succeq h2'$

and $(s1, h1), (s2, h2) \models J \wedge (s1, h1'), (s2, h2') \models J$

shows $h1' = h1 \wedge h2' = h2$

<proof>

definition *unary where*

unary J $\longleftrightarrow (\forall s h s' h'. (s, h), (s, h) \models J \wedge (s', h'), (s', h') \models J \longrightarrow (s, h), (s', h') \models J)$

lemma *unaryI*:

assumes $\bigwedge s1 h1 s2 h2. (s1, h1), (s1, h1) \models J \wedge (s2, h2), (s2, h2) \models J \implies (s1, h1), (s2, h2) \models J$

shows *unary J*

<proof>

lemma *unary-smallerI*:

assumes $\bigwedge \sigma1 \sigma2. \sigma1, \sigma1 \models J \wedge \sigma2, \sigma2 \models J \implies \sigma1, \sigma2 \models J$

shows *unary J*

<proof>

lemma *unaryE*:

assumes *unary J*

and $(s, h), (s, h) \models J \wedge (s', h'), (s', h') \models J$

shows $(s, h), (s', h') \models J$

<proof>

definition *entails* :: $(i, 'a, nat)$ *assertion* \Rightarrow $(i, 'a, nat)$ *assertion* \Rightarrow *bool* **where**

entails A B $\longleftrightarrow (\forall \sigma \sigma'. \sigma, \sigma' \models A \longrightarrow \sigma, \sigma' \models B)$

lemma *entailsI*:

assumes $\bigwedge x y. x, y \models A \implies x, y \models B$

shows *entails A B*

<proof>

lemma *sat-points-to*:

assumes $(s, h :: (i, 'a) \text{ heap}), (s, h) \models \text{PointsTo } a \ \pi \ e$

shows *get-fh* $h = [\text{edenot } a \ s \mapsto (\pi, \text{edenot } e \ s)]$

<proof>

lemma *unary-inv-then-view*:

assumes *unary J*

shows *unary* $(\text{View } f \ J \ e)$

<proof>

lemma *precise-inv-then-view*:

assumes *precise J*

shows *precise (View f J e)*

<proof>

fun *syntactic-unary* :: ('i, 'a, nat) *assertion* \Rightarrow *bool* **where**

syntactic-unary (Bool b) \longleftrightarrow *True*

| *syntactic-unary (And A B)* \longleftrightarrow *syntactic-unary A* \wedge *syntactic-unary B*

| *syntactic-unary (Star A B)* \longleftrightarrow *syntactic-unary A* \wedge *syntactic-unary B*

| *syntactic-unary (Low e)* \longleftrightarrow *False*

| *syntactic-unary Emp* \longleftrightarrow *True*

| *syntactic-unary (PointsTo v va vb)* \longleftrightarrow *True*

| *syntactic-unary (Exists x A)* \longleftrightarrow *syntactic-unary A*

| *syntactic-unary (SharedGuard - e)* \longleftrightarrow *True*

| *syntactic-unary (UniqueGuard - e)* \longleftrightarrow *True*

| *syntactic-unary (View view A e)* \longleftrightarrow *syntactic-unary A*

| *syntactic-unary (PreSharedGuards -)* \longleftrightarrow *False*

| *syntactic-unary (PreUniqueGuards -)* \longleftrightarrow *False*

| *syntactic-unary EmptyFullGuards* \longleftrightarrow *True*

| *syntactic-unary (LowExp x)* \longleftrightarrow *False*

| *syntactic-unary (Imp b A)* \longleftrightarrow *False*

lemma *syntactic-unary-implies-unary*:

assumes *syntactic-unary A*

shows *unary A*

<proof>

The following record defines resource contexts (Section 3.5).

record ('i, 'a, 'v) *single-context* =

view :: (*loc* \rightarrow *val*) \Rightarrow 'v

abstract-view :: 'v \Rightarrow 'v

saction :: 'v \Rightarrow 'a \Rightarrow 'v

uaction :: 'i \Rightarrow 'v \Rightarrow 'a \Rightarrow 'v

invariant :: ('i, 'a, 'v) *assertion*

type-synonym ('i, 'a, 'v) *cont* = ('i, 'a, 'v) *single-context* *option*

definition *no-guard-assertion* **where**

no-guard-assertion A \longleftrightarrow ($\forall s1 h1 s2 h2. (s1, h1), (s2, h2) \models A \longrightarrow$ *no-guard* *h1* \wedge *no-guard* *h2*)

Axiom that says that view only depends on the part of the heap described by the invariant inv.

definition *view-function-of-inv* :: ('i, 'a, nat) *single-context* \Rightarrow *bool* **where**

view-function-of-inv Γ \longleftrightarrow ($\forall (h :: ('i, 'a) \text{heap}) (h' :: ('i, 'a) \text{heap}) s. (s, h), (s, h) \models$ *invariant* $\Gamma \wedge (h' \succeq h)$

\longrightarrow *view* Γ (*normalize* (*get-fh* *h*)) = *view* Γ (*normalize* (*get-fh* *h'*)))

definition *wf-indexed-precondition* :: ('i ⇒ 'a ⇒ 'a ⇒ bool) ⇒ bool **where**
wf-indexed-precondition pre ↔ (∀ a b k. pre k a b → (pre k b a ∧ pre k a a))

definition *wf-precondition* :: ('a ⇒ 'a ⇒ bool) ⇒ bool **where**
wf-precondition pre ↔ (∀ a b. pre a b → (pre b a ∧ pre a a))

lemma *wf-precondition-rep-prec*:
assumes *wf-precondition pre*
shows *Rep-precondition (Abs-precondition pre) = pre*
 ⟨proof⟩

lemma *wf-indexed-precondition-rep-prec*:
assumes *wf-indexed-precondition pre*
shows *Rep-indexed-precondition (Abs-indexed-precondition pre) = pre*
 ⟨proof⟩

definition *LowView* **where**
LowView f A x = (Exists x (And (View f A (λs. s x)) (LowExp (Evar x))))

lemma *LowViewE*:
assumes (s, h), (s', h') ⊨ *LowView f A x*
and x ∉ fvA A
shows (s, h), (s', h') ⊨ A ∧ f (normalize (get-fh h)) = f (normalize (get-fh h'))
 ⟨proof⟩

lemma *LowViewI*:
assumes (s, h), (s', h') ⊨ A
and f (normalize (get-fh h)) = f (normalize (get-fh h'))
and x ∉ fvA A
shows (s, h), (s', h') ⊨ *LowView f A x*
 ⟨proof⟩

definition *disjoint* :: ('a set) ⇒ ('a set) ⇒ bool
where *disjoint h1 h2* = (h1 ∩ h2 = {})

definition *unambiguous* **where**
unambiguous A x ↔ (∀ s1 h1 s2 h2 v1 v2 v1' v2'. (s1(x := v1), h1), (s2(x := v2), h2) ⊨ A
 ∧ (s1(x := v1'), h1), (s2(x := v2'), h2) ⊨ A → v1 = v1' ∧ v2 = v2'))

definition *all-axioms* :: ('v ⇒ 'w) ⇒ ('v ⇒ 'a ⇒ 'v) ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('i
 ⇒ 'v ⇒ 'b ⇒ 'v) ⇒ ('i ⇒ 'b ⇒ 'b ⇒ bool) ⇒ bool **where**
all-axioms α sact spre uact upre ↔

— Every action's relational precondition is sufficient to preserve the low-ness of the abstract view of the resource value:

$$\begin{aligned} & (\forall v v' sarg sarg'. \alpha v = \alpha v' \wedge spre sarg sarg' \longrightarrow \alpha (sact v sarg) = \alpha (sact v' sarg')) \wedge \\ & (\forall v v' uarg uarg' i. \alpha v = \alpha v' \wedge upre i uarg uarg' \longrightarrow \alpha (uact i v uarg) = \alpha (uact i v' uarg')) \wedge \end{aligned}$$

$$\begin{aligned} & (\forall sarg sarg'. spre sarg sarg' \longrightarrow spre sarg' sarg') \wedge \\ & (\forall uarg uarg' i. upre i uarg uarg' \longrightarrow upre i uarg' uarg') \wedge \end{aligned}$$

— All relevant pairs of actions commute w.r.t. the abstract view:

$$\begin{aligned} & (\forall v v' sarg sarg'. \alpha v = \alpha v' \wedge spre sarg sarg \wedge spre sarg' sarg' \longrightarrow \alpha (sact (sact v sarg) sarg') = \alpha (sact (sact v' sarg') sarg)) \wedge \\ & (\forall v v' sarg uarg i. \alpha v = \alpha v' \wedge spre sarg sarg \wedge upre i uarg uarg \longrightarrow \alpha (sact (uact i v uarg) sarg) = \alpha (uact i (sact v' sarg) uarg)) \wedge \\ & (\forall v v' uarg uarg' i i'. i \neq i' \wedge \alpha v = \alpha v' \wedge upre i uarg uarg \wedge upre i' uarg' uarg' \\ & \longrightarrow \alpha (uact i' (uact i v uarg) uarg') = \alpha (uact i (uact i' v' uarg') uarg)) \end{aligned}$$

3.2 Rules of the Logic

inductive *CommCSL* :: ('i, 'a, nat) cont \Rightarrow ('i, 'a, nat) assertion \Rightarrow cmd \Rightarrow ('i, 'a, nat) assertion \Rightarrow bool

($\langle \cdot \vdash \{-\} - \{-\} \rangle$ [51,0,0] 81) **where**

$$\begin{aligned} & \text{RuleSkip: } \Delta \vdash \{P\} \text{ Cskip } \{P\} \\ | \text{RuleAssign: } \llbracket \bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies x \notin \text{fvA } (\text{invariant } \Gamma) ; \text{collect-existentials } \\ & P \cap \text{fvE } E = \{\} \rrbracket \implies \Delta \vdash \{\text{subA } x E P\} \text{ Cassign } x E \{P\} \\ | \text{RuleNew: } \llbracket x \notin \text{fvE } E ; \bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies x \notin \text{fvA } (\text{invariant } \Gamma) \wedge \text{view-function-of-inv } \\ & \Gamma \rrbracket \implies \Delta \vdash \{\text{Emp}\} \text{ Calloc } x E \{\text{PointsTo } (Evar x) \text{ pwrite } E\} \\ | \text{RuleWrite: } \llbracket \bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies \text{view-function-of-inv } \Gamma ; v \notin \text{fvE } \text{loc} \rrbracket \\ & \implies \Delta \vdash \{\text{Exists } v (\text{PointsTo } \text{loc } \text{pwrite } (Evar v))\} \text{ Cwrite } \text{loc } E \{\text{PointsTo } \text{loc } \\ & \text{pwrite } E\} \\ | \llbracket \bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies x \notin \text{fvA } (\text{invariant } \Gamma) \wedge \text{view-function-of-inv } \Gamma ; x \notin \text{fvE } \\ & E \cup \text{fvE } e \rrbracket \implies \\ & \Delta \vdash \{\text{PointsTo } E \pi e\} \text{ Cread } x E \{\text{And } (\text{PointsTo } E \pi e) (\text{Bool } (\text{Beq } (Evar x) \\ & e))\} \\ | \text{RuleShare: } \llbracket \Gamma = () \text{ view} = f, \text{abstract-view} = \alpha, \text{saction} = \text{sact}, \text{uaction} = \text{uact}, \\ & \text{invariant} = J \rrbracket ; \text{all-axioms } \alpha \text{ sact spre uact upre} ; \\ & \text{Some } \Gamma \vdash \{\text{Star } P \text{ EmptyFullGuards}\} C \{\text{Star } Q (\text{And } (\text{PreSharedGuards } (\text{Abs-precondition } \\ & \text{spre})) (\text{PreUniqueGuards } (\text{Abs-indexed-precondition } \text{upre})))\}; \\ & \text{view-function-of-inv } \Gamma ; \text{unary } J ; \text{precise } J ; \text{wf-indexed-precondition } \text{upre} ; \\ & \text{wf-precondition } \text{spre} ; x \notin \text{fvA } J ; \\ & \text{no-guard-assertion } (\text{Star } P (\text{LowView } (\alpha \circ f) J x)) \rrbracket \implies \text{None} \vdash \{\text{Star } P \\ & (\text{LowView } (\alpha \circ f) J x)\} C \{\text{Star } Q (\text{LowView } (\alpha \circ f) J x)\} \\ | \text{RuleAtomicUnique: } \llbracket \Gamma = () \text{ view} = f, \text{abstract-view} = \alpha, \text{saction} = \text{sact}, \text{uaction} \\ & = \text{uact}, \text{invariant} = J \rrbracket ; \\ & \text{no-guard-assertion } P ; \text{no-guard-assertion } Q ; \\ & \text{None} \vdash \{\text{Star } P (\text{View } f J (\lambda s. s x))\} C \{\text{Star } Q (\text{View } f J (\lambda s. \text{uact } \text{index } (s \\ & x) (\text{map-to-arg } (s \text{ uarg})))\} ; \end{aligned}$$

*precise J ; unary J ; view-function-of-inv Γ ; $x \notin \text{fvC } C \cup \text{fvA } P \cup \text{fvA } Q \cup \text{fvA } J ; \text{uarg} \notin \text{fvC } C ;$
 $l \notin \text{fvC } C ; x \notin \text{fvS } (\lambda s. \text{map-to-list } (s l)) ; x \notin \text{fvS } (\lambda s. \text{map-to-arg } (s \text{uarg}) \# \text{map-to-list } (s l))$]]
 $\implies \text{Some } \Gamma \vdash \{ \text{Star } P (\text{UniqueGuard index } (\lambda s. \text{map-to-list } (s l))) \} \text{Catomic } C \{ \text{Star } Q (\text{UniqueGuard index } (\lambda s. \text{map-to-arg } (s \text{uarg}) \# \text{map-to-list } (s l))) \}$
| *RuleAtomicShared:* [$\Gamma = () \text{ view} = f, \text{abstract-view} = \alpha, \text{saction} = \text{sact}, \text{uaction} = \text{uact}, \text{invariant} = J$) ; *no-guard-assertion* $P ; \text{no-guard-assertion } Q ;$
 $\text{None} \vdash \{ \text{Star } P (\text{View } f J (\lambda s. s x)) \} C \{ \text{Star } Q (\text{View } f J (\lambda s. \text{sact } (s x) (\text{map-to-arg } (s \text{sarg})))) \} ;$
*precise J ; unary J ; view-function-of-inv Γ ; $x \notin \text{fvC } C \cup \text{fvA } P \cup \text{fvA } Q \cup \text{fvA } J ; \text{sarg} \notin \text{fvC } C ;$
 $ms \notin \text{fvC } C ; x \notin \text{fvS } (\lambda s. \text{map-to-multiset } (s ms)) ; x \notin \text{fvS } (\lambda s. \{ \# \text{map-to-arg } (s \text{sarg}) \# \} + \text{map-to-multiset } (s ms))$]]
 $\implies \text{Some } \Gamma \vdash \{ \text{Star } P (\text{SharedGuard } \pi (\lambda s. \text{map-to-multiset } (s ms))) \} \text{Catomic } C \{ \text{Star } Q (\text{SharedGuard } \pi (\lambda s. \{ \# \text{map-to-arg } (s \text{sarg}) \# \} + \text{map-to-multiset } (s ms))) \}$
| *RulePar:* [$\Delta \vdash \{ P1 \} C1 \{ Q1 \} ; \Delta \vdash \{ P2 \} C2 \{ Q2 \} ; \text{disjoint } (\text{fvA } P1 \cup \text{fvC } C1 \cup \text{fvA } Q1) (\text{wrC } C2) ;$
 $\text{disjoint } (\text{fvA } P2 \cup \text{fvC } C2 \cup \text{fvA } Q2) (\text{wrC } C1) ; \bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies \text{disjoint } (\text{fvA } (\text{invariant } \Gamma)) (\text{wrC } C2) ;$
 $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies \text{disjoint } (\text{fvA } (\text{invariant } \Gamma)) (\text{wrC } C1) ; \text{precise } P1 \vee \text{precise } P2$]]
 $\implies \Delta \vdash \{ \text{Star } P1 P2 \} \text{Cpar } C1 C2 \{ \text{Star } Q1 Q2 \}$
| *RuleIf1:* [$\Delta \vdash \{ \text{And } P (\text{Bool } b) \} C1 \{ Q \} ; \Delta \vdash \{ \text{And } P (\text{Bool } (\text{Bnot } b)) \} C2 \{ Q \}$]]
 $\implies \Delta \vdash \{ \text{And } P (\text{Low } b) \} \text{Cif } b C1 C2 \{ Q \}$
| *RuleIf2:* [$\Delta \vdash \{ \text{And } P (\text{Bool } b) \} C1 \{ Q \} ; \Delta \vdash \{ \text{And } P (\text{Bool } (\text{Bnot } b)) \} C2 \{ Q \} ; \text{unary } Q$]]
 $\implies \Delta \vdash \{ P \} \text{Cif } b C1 C2 \{ Q \}$
| *RuleSeq:* [$\Delta \vdash \{ P \} C1 \{ R \} ; \Delta \vdash \{ R \} C2 \{ Q \}$]] $\implies \Delta \vdash \{ P \} \text{Cseq } C1 C2 \{ Q \}$
| *RuleFrame:* [$\Delta \vdash \{ P \} C \{ Q \} ; \text{disjoint } (\text{fvA } R) (\text{wrC } C) ; \text{precise } P \vee \text{precise } R$]]
 $\implies \Delta \vdash \{ \text{Star } P R \} C \{ \text{Star } Q R \}$
| *RuleCons:* [$\Delta \vdash \{ P' \} C \{ Q' \} ; \text{entails } P P' ; \text{entails } Q' Q$]] $\implies \Delta \vdash \{ P \} C \{ Q \}$
| *RuleExists:* [$\Delta \vdash \{ P \} C \{ Q \} ; x \notin \text{fvC } C ; \bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies x \notin \text{fvA } (\text{invariant } \Gamma) ; \text{unambiguous } P x$]]
 $\implies \Delta \vdash \{ \text{Exists } x P \} C \{ \text{Exists } x Q \}$
| *RuleWhile1:* $\Delta \vdash \{ \text{And } I (\text{Bool } b) \} C \{ \text{And } I (\text{Low } b) \} \implies \Delta \vdash \{ \text{And } I (\text{Low } b) \} \text{Cwhile } b C \{ \text{And } I (\text{Bool } (\text{Bnot } b)) \}$
| *RuleWhile2:* [$\text{unary } I ; \Delta \vdash \{ \text{And } I (\text{Bool } b) \} C \{ I \}$]] $\implies \Delta \vdash \{ I \} \text{Cwhile } b C \{ \text{And } I (\text{Bool } (\text{Bnot } b)) \}$**

end

4 Soundness of CommCSL

4.1 Abstract Commutativity

In this file, we prove lemma 4.2 from the paper: Essentially, conditions (1)-(4) from Section 2 are sufficient to ensure that the abstraction of the final shared value is low.

theory *AbstractCommutativity*

imports *Main CommCSL HOL-Library.Multiset*

begin

datatype (*'i, 'a, 'b*) *action* = *Shared (get-s: 'a) | Unique (get-i: 'i) (get-u: 'b)*

We consider a family of unique actions indexed by the type *'i*

lemma *sabstract*:

assumes *all-axioms* α *sact spre uact upre*

shows $\alpha v = \alpha v' \wedge spre\ sarg\ sarg' \implies \alpha (sact\ v\ sarg) = \alpha (sact\ v'\ sarg')$

<proof>

lemma *uabstract*:

assumes *all-axioms* α *sact spre uact upre*

shows $\alpha v = \alpha v' \wedge upre\ i\ uarg\ uarg' \implies \alpha (uact\ i\ v\ uarg) = \alpha (uact\ i\ v'\ uarg')$

<proof>

lemma *spre-refl*:

assumes *all-axioms* α *sact spre uact upre*

shows *spre sarg sarg' $\implies spre\ sarg'\ sarg'$*

<proof>

lemma *upre-refl*:

assumes *all-axioms* α *sact spre uact upre*

shows *upre i uarg uarg' $\implies upre\ i\ uarg'\ uarg'$*

<proof>

lemma *ss-com*:

assumes *all-axioms* α *sact spre uact upre*

shows $\alpha v = \alpha v' \implies spre\ sarg\ sarg' \wedge spre\ sarg'\ sarg' \implies \alpha (sact\ (sact\ v\ sarg)\ sarg') = \alpha (sact\ (sact\ v'\ sarg')\ sarg)$

<proof>

lemma *su-com*:

assumes *all-axioms* α *sact spre uact upre*

shows $\alpha v = \alpha v' \implies spre\ sarg\ sarg' \wedge upre\ i\ uarg\ uarg' \implies \alpha (sact\ (uact\ i\ v\ uarg)\ sarg) = \alpha (uact\ i\ (sact\ v'\ sarg)\ uarg)$

<proof>

lemma *uu-com*:

assumes *all-axioms* α *sact spre uact upre*

and $i \neq i'$

and $\alpha v = \alpha v'$
and $\text{upre } i' \text{ uarg}' \text{ uarg}'$
and $\text{upre } i \text{ uarg} \text{ uarg}$
shows $\alpha (\text{uact } i' (\text{uact } i v \text{ uarg}) \text{ uarg}') = \alpha (\text{uact } i (\text{uact } i' v' \text{ uarg}') \text{ uarg})$
 <proof>

definition *PRE-shared* :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a multiset \Rightarrow 'a multiset \Rightarrow bool
where

PRE-shared spre sargs sargs' $\longleftrightarrow (\exists \text{ms. image-mset fst ms} = \text{sargs} \wedge \text{image-mset snd ms} = \text{sargs}' \wedge (\forall x \in \# \text{ms. spre (fst } x) (\text{snd } x)))$

lemma *PRE-shared-same-size*:

assumes *PRE-shared spre sargs sargs'*
shows $\text{size sargs} = \text{size sargs}'$
 <proof>

definition *is-Unique* :: ('i, 'a, 'b) action \Rightarrow bool **where**
is-Unique a $\longleftrightarrow \neg \text{is-Shared } a$

definition *is-Unique-i* :: 'i \Rightarrow ('i, 'a, 'b) action \Rightarrow bool **where**
is-Unique-i i a $\longleftrightarrow \text{is-Unique } a \wedge \text{get-}i \text{ } a = i$

The following definition expresses that a sequence of actions corresponds to some interleaving of a multiset of shared actions and a family of sequences of unique actions, by projecting the sequence of actions on each type of action.

definition *possible-sequence* :: 'a multiset \Rightarrow ('i \Rightarrow 'b list) \Rightarrow ('i, 'a, 'b) action list \Rightarrow bool **where**

possible-sequence sargs uargs s $\longleftrightarrow ((\forall i. \text{uargs } i = \text{map get-}u (\text{filter } (\text{is-Unique-}i \text{ } i) \text{ } s)) \wedge \text{sargs} = \text{image-mset get-}s (\text{filter-mset is-Shared } (\text{mset } s)))$

lemma *possible-sequenceI*:

assumes $\bigwedge i. \text{uargs } i = \text{map get-}u (\text{filter } (\text{is-Unique-}i \text{ } i) \text{ } s)$
and $\text{sargs} = \text{image-mset get-}s (\text{filter-mset is-Shared } (\text{mset } s))$
shows *possible-sequence sargs uargs s*
 <proof>

fun *remove-at-index* :: nat \Rightarrow 'd list \Rightarrow 'd list **where**

$\text{remove-at-index } - \ [] = []$
 $\text{remove-at-index } 0 \ (x \# \text{xs}) = \text{xs}$
 $\text{remove-at-index } (\text{Suc } n) \ (x \# \text{xs}) = x \# (\text{remove-at-index } n \ \text{xs})$

lemma *remove-at-index*:

assumes $n < \text{length } l$
shows $\text{length } (\text{remove-at-index } n \ l) = \text{length } l - 1$
and $i \geq 0 \wedge i < n \implies \text{remove-at-index } n \ l ! i = l ! i$
and $i \geq n \wedge i < \text{length } l - 1 \implies \text{remove-at-index } n \ l ! i = l ! (i + 1)$
 <proof>

fun *insert-at* :: *nat* \Rightarrow '*d* \Rightarrow '*d list* \Rightarrow '*d list* **where**
insert-at 0 *x l* = *x # l*
| *insert-at* - *x []* = [*x*]
| *insert-at* (*Suc n*) *x* (*h # xs*) = *h # (insert-at n x xs)*

lemma *insert-at-index*:
assumes $n \leq \text{length } l$
shows $\text{length } (\text{insert-at } n \ x \ l) = \text{length } l + 1$
and $i \geq 0 \wedge i < n \implies \text{insert-at } n \ x \ l ! i = l ! i$
and $n \geq 0 \implies \text{insert-at } n \ x \ l ! n = x$
and $i > n \wedge i < \text{length } l + 1 \implies \text{insert-at } n \ x \ l ! i = l ! (i - 1)$
⟨*proof*⟩

lemma *list-ext*:
assumes $\text{length } a = \text{length } b$
and $\bigwedge i. i \geq 0 \wedge i < \text{length } a \implies a ! i = b ! i$
shows $a = b$
⟨*proof*⟩

lemma *mset-remove-index*:
assumes $i \geq 0 \wedge i < \text{length } l$
shows $\text{mset } l = \text{mset } (\text{remove-at-index } i \ l) + \{\# \ l ! i \ \#\}$
⟨*proof*⟩

lemma *filter-remove*:
assumes $k \geq 0 \wedge k < \text{length } s$
and $\neg P \ (s ! k)$
shows $\text{filter } P \ (\text{remove-at-index } k \ s) = \text{filter } P \ s$
⟨*proof*⟩

lemma *exists-index-in-sequence-shared*:
assumes $a \in \# \ \text{sargs}$
and *possible-sequence sargs uargs s*
shows $\exists i. i \geq 0 \wedge i < \text{length } s \wedge s ! i = \text{Shared } a \wedge \text{possible-sequence } (\text{sargs} - \{\# \ a \ \#\}) \ uargs \ (\text{remove-at-index } i \ s)$
⟨*proof*⟩

lemma *head-possible-exists-first-unique*:
assumes $a = \text{hd } (uargs \ j)$
and $uargs \ j \neq []$
and *possible-sequence sargs uargs s*
shows $\exists i. i \geq 0 \wedge i < \text{length } s \wedge s ! i = \text{Unique } j \ a \wedge (\forall k. k \geq 0 \wedge k < i \longrightarrow \neg \text{is-Unique-i } j \ (s ! k))$
⟨*proof*⟩

lemma *remove-at-index-filter*:
assumes $i \geq 0 \wedge i < \text{length } s \wedge P \ (s ! i)$
and $\bigwedge j. j \geq 0 \wedge j < i \implies \neg P \ (s ! j)$

shows $tl (map\ get-u (filter\ P\ s)) = map\ get-u (filter\ P (remove-at-index\ i\ s))$
 ⟨proof⟩

definition *tail-kth where*

$tail-kth\ uargs\ k = uargs(k := tl (uargs\ k))$

lemma *exists-index-in-sequence-unique:*

assumes $a = hd (uargs\ k)$

and $uargs\ k \neq []$

and *possible-sequence sargs uargs s*

shows $\exists i. i \geq 0 \wedge i < length\ s \wedge s ! i = Unique\ k\ a \wedge possible-sequence\ sargs$
 $(tail-kth\ uargs\ k) (remove-at-index\ i\ s)$

$\wedge (\forall j. j \geq 0 \wedge j < i \longrightarrow \neg is-Unique-i\ k (s ! j))$

⟨proof⟩

lemma *possible-sequence-where-is-unique:*

assumes *possible-sequence sargs uargs (Unique k a # s)*

shows $a = hd (uargs\ k)$

⟨proof⟩

lemma *possible-sequence-where-is-shared:*

assumes *possible-sequence sargs uargs (Shared a # s)*

shows $a \in \# sargs$

⟨proof⟩

lemma *PRE-unique-tII:*

assumes *PRE-unique upre qa qb*

and *upre ta tb*

shows *PRE-unique upre (ta # qa) (tb # qb)*

⟨proof⟩

fun *abstract-pre* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('i \Rightarrow 'b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('i, 'a, 'b)\ action$
 $\Rightarrow ('i, 'a, 'b)\ action \Rightarrow bool$ **where**

$abstract-pre\ spre\ upre (Shared\ sarg) (Shared\ sarg') \longleftrightarrow spre\ sarg\ sarg'$

$| abstract-pre\ spre\ upre (Unique\ k\ uarg) (Unique\ k'\ uarg') \longleftrightarrow k = k' \wedge upre\ k$
 $uarg\ uarg'$

$| abstract-pre\ spre\ upre\ -\ - \longleftrightarrow False$

definition *PRE-sequence* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('i \Rightarrow 'b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('i,$
 $'a, 'b)\ action\ list \Rightarrow ('i, 'a, 'b)\ action\ list \Rightarrow bool$ **where**

$PRE-sequence\ spre\ upre\ s\ s' \longleftrightarrow length\ s = length\ s' \wedge (\forall i. i \geq 0 \wedge i < length$
 $s \longrightarrow abstract-pre\ spre\ upre (s ! i) (s' ! i))$

lemma *PRE-sequenceE:*

assumes *PRE-sequence spre upre s s'*

and $i \geq 0 \wedge i < length\ s$

shows *abstract-pre spre upre (s ! i) (s' ! i)*

⟨proof⟩

lemma *PRE-sequenceI*:

assumes $\text{length } s = \text{length } s'$

and $\bigwedge i. i \geq 0 \wedge i < \text{length } s \implies \text{abstract-pre spre upre } (s ! i) (s' ! i)$

shows *PRE-sequence spre upre s s'*

<proof>

lemma *PRE-sequenceI-rec*:

assumes *PRE-sequence spre upre s s'*

and *abstract-pre spre upre a b*

shows *PRE-sequence spre upre (a # s) (b # s')*

<proof>

lemma *PRE-sequenceE-rec*:

assumes *PRE-sequence spre upre (a # s) (b # s')*

shows *PRE-sequence spre upre s s'*

and *abstract-pre spre upre a b*

<proof>

fun *compute* :: $('v \Rightarrow 'a \Rightarrow 'v) \Rightarrow ('i \Rightarrow 'v \Rightarrow 'b \Rightarrow 'v) \Rightarrow 'v \Rightarrow ('i, 'a, 'b) \text{ action}$
list $\Rightarrow 'v$ **where**

compute *sact uact v0* [] = *v0*

| *compute* *sact uact v0* (*Shared sarg # s*) = *sact (compute sact uact v0 s) sarg*

| *compute* *sact uact v0* (*Unique k uarg # s*) = *uact k (compute sact uact v0 s) uarg*

lemma *obtain-other-elem-ms*:

assumes *PRE-shared spre sargs sargs'*

and $sarg \in \# sargs$

shows $\exists sarg'. sarg' \in \# sargs' \wedge \text{spre sarg sarg}' \wedge \text{PRE-shared spre } (sargs - \{\# sarg \# \}) (sargs' - \{\# sarg' \# \})$

<proof>

lemma *exists-aligned-sequence*:

assumes *possible-sequence sargs uargs s*

and *possible-sequence sargs' uargs' s'*

and *PRE-shared spre sargs sargs'*

and $\bigwedge k. \text{PRE-unique } (\text{upre } k) (\text{uargs } k) (\text{uargs}' k)$

shows $\exists s''. \text{possible-sequence } sargs' uargs' s'' \wedge \text{PRE-sequence spre upre } s s''$

<proof>

lemma *insert-remove-same-list*:

assumes $k \geq 0 \wedge k < \text{length } s$

and $s ! k = x$

shows $s = \text{insert-at } k x (\text{remove-at-index } k s)$

<proof>

lemma *swap-works*:

assumes $\text{length } s = \text{length } s'$
and $k < \text{length } s - 1$
and $\bigwedge i. i \geq 0 \wedge i < \text{length } s \wedge i \neq k \wedge i \neq k + 1 \implies s ! i = s' ! i$
and $s ! k = s' ! (k + 1)$
and $s' ! k = s ! (k + 1)$
and *PRE-sequence spre upre s s*
and $\alpha v0 = \alpha v0'$
and $\neg (\exists k'. \text{is-Unique-}i\ k' (s ! k) \wedge \text{is-Unique-}i\ k' (s ! (k + 1)))$
and *all-axioms α sact spre uact upre*
shows $\alpha (\text{compute sact uact } v0\ s) = \alpha (\text{compute sact uact } v0'\ s')$ (**is** ?A = ?B)
(*proof*)

lemma *mset-remove*:

assumes $k \geq 0 \wedge k < \text{length } s$
shows $\text{mset } s = \text{mset } (\text{remove-at-index } k\ s) + \{\# s ! k \#\}$
(*proof*)

lemma *abstract-pre-refl*:

assumes *abstract-pre spre upre a b*
and *all-axioms α sact spre uact upre*
shows *abstract-pre spre upre b b*
(*proof*)

lemma *PRE-sequence-refl*:

assumes *PRE-sequence spre upre s s'*
and *all-axioms α sact spre uact upre*
shows *PRE-sequence spre upre s' s'*
(*proof*)

lemma *PRE-sequence-removes*:

assumes *PRE-sequence spre upre s s*
shows *PRE-sequence spre upre (remove-at-index n s) (remove-at-index n s)*
(*proof*)

lemma *PRE-sequence-insert*:

assumes *abstract-pre spre upre x x*
and *PRE-sequence spre upre s s*
shows *PRE-sequence spre upre (insert-at n x s) (insert-at n x s)*
(*proof*)

lemma *empty-possible-sequence*:

assumes *possible-sequence sargs uargs []*
and *possible-sequence sargs uargs s'*
shows $s' = []$
(*proof*)

lemma *it-all-commutes*:

assumes *possible-sequence sargs uargs s*

and *possible-sequence* $sargs\ uargs\ s'$
and $\alpha\ v0 = \alpha\ v0'$
and *PRE-sequence* $spre\ upre\ s\ s$
and *PRE-sequence* $spre\ upre\ s'\ s'$
and *all-axioms* $\alpha\ sact\ spre\ uact\ upre$
shows $\alpha\ (compute\ sact\ uact\ v0\ s) = \alpha\ (compute\ sact\ uact\ v0'\ s')$
<proof>

lemma *PRE-sequence-same-abstract*:
assumes *PRE-sequence* $spre\ upre\ s\ s'$
and $\alpha\ v0 = \alpha\ v0'$
and *all-axioms* $\alpha\ sact\ spre\ uact\ upre$
shows $\alpha\ (compute\ sact\ uact\ v0\ s) = \alpha\ (compute\ sact\ uact\ v0'\ s')$
<proof>

lemma *simple-possible-PRE-seq*:
assumes *possible-sequence* $sargs\ uargs\ s$
and *possible-sequence* $sargs'\ uargs'\ s'$
and *PRE-shared* $spre\ sargs\ sargs'$
and $\bigwedge k. PRE\text{-unique}\ (upre\ k)\ (uargs\ k)\ (uargs'\ k)$
and *all-axioms* $\alpha\ sact\ spre\ uact\ upre$
shows *PRE-sequence* $spre\ upre\ s'\ s'$
<proof>

lemma *main-lemma*:
assumes *possible-sequence* $sargs\ uargs\ s$
and *possible-sequence* $sargs'\ uargs'\ s'$

and *PRE-shared* $spre\ sargs\ sargs'$
and $\bigwedge k. PRE\text{-unique}\ (upre\ k)\ (uargs\ k)\ (uargs'\ k)$

and $\alpha\ v0 = \alpha\ v0'$
and *all-axioms* $\alpha\ sact\ spre\ uact\ upre$

shows $\alpha\ (compute\ sact\ uact\ v0\ s) = \alpha\ (compute\ sact\ uact\ v0'\ s')$
<proof>

The following inductive predicate captures all possible final values that can be reached with some interleaving of the actions described a multiset and a family of sequences of actions.

inductive *reachable-value* :: $('v \Rightarrow 'a \Rightarrow 'v) \Rightarrow ('i \Rightarrow 'v \Rightarrow 'b \Rightarrow 'v) \Rightarrow 'v \Rightarrow 'a$
 $multiset \Rightarrow ('i \Rightarrow 'b\ list) \Rightarrow 'v \Rightarrow bool$ **where**
Self: *reachable-value* $sact\ uact\ v0\ \{\#\}$ $(\lambda k. \square)\ v0$
SharedStep: *reachable-value* $sact\ uact\ v0\ sargs\ uargs\ v1 \implies reachable\text{-value}\ sact$
 $uact\ v0\ (sargs + \{\#\ sarg\ \#\})\ uargs\ (sact\ v1\ sarg)$
UniqueStep: *reachable-value* $sact\ uact\ v0\ sargs\ uargs\ v1 \implies reachable\text{-value}\ sact$
 $uact\ v0\ sargs\ (uargs(k := uarg\ \#\ uargs\ k))\ (uact\ k\ v1\ uarg)$

lemma *reachable-then-possible-sequence-and-compute*:

assumes *reachable-value sact uact v0 sargs uargs v1*
shows $\exists s. \text{possible-sequence sargs uargs s} \wedge v1 = \text{compute sact uact v0 s}$
<proof>

lemma *PRE-shared-simpler-implies:*
assumes *PRE-shared-simpler spre a b*
shows *PRE-shared spre a b*
<proof>

The following theorem corresponds to Lemma 4.2 in the paper.

theorem *main-result:*
assumes *reachable-value sact uact v0 sargs uargs v*
and *reachable-value sact uact v0' sargs' uargs' v'*
and *PRE-shared-simpler spre sargs sargs'*
and $\bigwedge k. \text{PRE-unique (upre k) (uargs k) (uargs' k)}$
and $\alpha v0 = \alpha v0'$
and *all-axioms α sact spre uact upre*
shows $\alpha v = \alpha v'$
<proof>

end

4.2 Consistency

In this file, we define several notions and prove many lemmas about guard states, which are useful to prove that the rules of the logic are sound.

theory *Guards*
imports *StateModel CommCSL AbstractCommutativity*
begin

A state is "consistent" iff: 1. All its permissions are full 2. Has unique guards iff has shared guard 3. The values in the fractional heaps are "reachable" wrt to the sequence and multiset of actions 4. Has exactly guards for the names in "scope"

definition *reachable* :: $(i, 'a, 'v) \text{single-context} \Rightarrow 'v \Rightarrow (i, 'a) \text{heap} \Rightarrow \text{bool}$ **where**
 $\text{reachable scont v0 h} \iff (\forall \text{sargs uargs. get-gs h} = \text{Some (pwrite, sargs)} \wedge (\forall k. \text{get-gu h k} = \text{Some (uargs k)}))$
 $\implies \text{reachable-value (saction scont) (uaction scont) v0 sargs uargs (view scont (normalize (get-fh h)))}$

lemma *reachableI:*
assumes $\bigwedge \text{sargs uargs. get-gs h} = \text{Some (pwrite, sargs)} \wedge (\forall k. \text{get-gu h k} = \text{Some (uargs k)})$
 $\implies \text{reachable-value (saction scont) (uaction scont) v0 sargs uargs (view scont (normalize (get-fh h)))}$
shows *reachable scont v0 h*
<proof>

lemma *reachableE*:

assumes *reachable scont v0 h*
and *get-gs h = Some (pwrite, sargs)*
and $\bigwedge k. \text{get-gu } h \ k = \text{Some } (\text{uargs } k)$
shows *reachable-value (saction scont) (uaction scont) v0 sargs uargs (view scont (normalize (get-fh h)))*
<proof>

definition *all-guards* :: $(\text{'i}, \text{'a}) \text{ heap} \Rightarrow \text{bool}$ **where**

all-guards h $\longleftrightarrow (\exists v. \text{get-gs } h = \text{Some } (\text{pwrite}, v)) \wedge (\forall k. \text{get-gu } h \ k \neq \text{None})$

lemma *no-guardI*:

assumes *get-gs h = None*
and $\bigwedge k. \text{get-gu } h \ k = \text{None}$
shows *no-guard h*
<proof>

definition *semi-consistent* :: $(\text{'i}, \text{'a}, \text{'v}) \text{ single-context} \Rightarrow \text{'v} \Rightarrow (\text{'i}, \text{'a}) \text{ heap} \Rightarrow \text{bool}$
where

semi-consistent $\Gamma \ v0 \ h \longleftrightarrow \text{all-guards } h \wedge \text{reachable } \Gamma \ v0 \ h$

lemma *semi-consistentE*:

assumes *semi-consistent* $\Gamma \ v0 \ h$
shows $\exists \text{sargs } \text{uargs}. \text{get-gs } h = \text{Some } (\text{pwrite}, \text{sargs}) \wedge (\forall k. \text{get-gu } h \ k = \text{Some } (\text{uargs } k))$
 $\wedge \text{reachable-value } (\text{saction } \Gamma) (\text{uaction } \Gamma) \ v0 \ \text{sargs } \text{uargs} (\text{view } \Gamma (\text{normalize } (\text{get-fh } h)))$
<proof>

lemma *semi-consistentI*:

assumes *all-guards h*
and *reachable* $\Gamma \ v0 \ h$
shows *semi-consistent* $\Gamma \ v0 \ h$
<proof>

lemma *no-guard-then-smaller-same*:

assumes *Some h = Some a \oplus Some b*
and *no-guard h*
shows *no-guard a*
<proof>

lemma *all-guardsI*:

assumes $\bigwedge k. \text{get-gu } h \ k \neq \text{None}$
and $\exists v. \text{get-gs } h = \text{Some } (\text{pwrite}, v)$
shows *all-guards h*
<proof>

lemma *all-guards-same*:

assumes *all-guards a*

and $\text{Some } h = \text{Some } a \oplus \text{Some } b$
shows *all-guards* h
 ⟨*proof*⟩

definition *empty-unique* **where**
empty-unique $- = \text{None}$

definition *remove-guards* $:: ('i, 'a) \text{ heap} \Rightarrow ('i, 'a) \text{ heap}$ **where**
remove-guards $h = (\text{get-fh } h, \text{None}, \text{empty-unique})$

lemma *remove-guards-smaller*:
 $h \succeq \text{remove-guards } h$
 ⟨*proof*⟩

lemma *no-guard-remove*:
assumes $\text{Some } a = \text{Some } b \oplus \text{Some } c$
and *no-guard* c
shows $\text{get-gs } a = \text{get-gs } b$
and $\text{get-gu } a = \text{get-gu } b$
 ⟨*proof*⟩

lemma *full-guard-comp-then-no*:
assumes $a \#\# b$
and *all-guards* a
shows *no-guard* b
 ⟨*proof*⟩

lemma *sum-of-no-guards*:
assumes *no-guard* a
and *no-guard* b
and $\text{Some } x = \text{Some } a \oplus \text{Some } b$
shows *no-guard* x
 ⟨*proof*⟩

lemma *no-guard-remove-guards*:
no-guard $(\text{remove-guards } h)$
 ⟨*proof*⟩

lemma *get-fh-remove-guards*:
 $\text{get-fh } (\text{remove-guards } h) = \text{get-fh } h$
 ⟨*proof*⟩

definition *pair-sat* $:: (\text{store} \times ('i, 'a) \text{ heap}) \text{ set} \Rightarrow (\text{store} \times ('i, 'a) \text{ heap}) \text{ set} \Rightarrow$
 $('i, 'a, \text{nat}) \text{ assertion} \Rightarrow \text{bool}$ **where**
pair-sat $S S' Q \iff (\forall \sigma \sigma'. \sigma \in S \wedge \sigma' \in S' \longrightarrow \sigma, \sigma' \models Q)$

lemma *pair-satI*:
assumes $\bigwedge s h s' h'. (s, h) \in S \wedge (s', h') \in S' \implies (s, h), (s', h') \models Q$
shows *pair-sat* $S S' Q$

<proof>

lemma *pair-sat-smallerI*:

assumes $\bigwedge \sigma \sigma'. \sigma \in S \wedge \sigma' \in S' \implies \sigma, \sigma' \models Q$

shows *pair-sat* $S S' Q$

<proof>

lemma *pair-satE*:

assumes *pair-sat* $S S' Q$

and $(s, h) \in S \wedge (s', h') \in S'$

shows $(s, h), (s', h') \models Q$

<proof>

definition *add-states* :: $(store \times ('i, 'a) heap) set \implies (store \times ('i, 'a) heap) set \implies (store \times ('i, 'a) heap) set$ **where**

add-states $S1 S2 = \{(s, H) \mid s H h1 h2. \text{Some } H = \text{Some } h1 \oplus \text{Some } h2 \wedge (s, h1) \in S1 \wedge (s, h2) \in S2\}$

lemma *add-states-sat-star*:

assumes *pair-sat* $SA SA' A$

and *pair-sat* $SB SB' B$

shows *pair-sat* $(\text{add-states } SA SB) (\text{add-states } SA' SB') (\text{Star } A B)$

<proof>

lemma *add-states-subset*:

assumes $S1 \subseteq S1'$

shows *add-states* $S1 S2 \subseteq \text{add-states } S1' S2$

<proof>

lemma *add-states-comm*:

add-states $S1 S2 = \text{add-states } S2 S1$

<proof>

The following lemma is the reason why we require many assertions to be precise in the logic.

lemma *magic-lemma*:

assumes $\text{Some } x1 = \text{Some } a1 \oplus \text{Some } j1$

and $\text{Some } x2 = \text{Some } a2 \oplus \text{Some } j2$

and $(s1, x1), (s2, x2) \models \text{Star } A J$

and $(s1, j1), (s2, j2) \models J$

and *precise* J

shows $(s1, a1), (s2, a2) \models A$

<proof>

lemma *full-no-guard-same-normalize*:

assumes *full-ownership* $(\text{get-fh } h) \wedge \text{no-guard } h$

and *full-ownership* $(\text{get-fh } h') \wedge \text{no-guard } h'$

and *normalize* $(\text{get-fh } h) = \text{normalize } (\text{get-fh } h')$

shows $h = h'$
<proof>

lemma *get-fh-same-then-remove-guards-same:*
assumes $\text{get-fh } a = \text{get-fh } b$
shows $\text{remove-guards } a = \text{remove-guards } b$
<proof>

lemma *remove-guards-sum:*
assumes $\text{Some } x = \text{Some } a \oplus \text{Some } b$
shows $\text{Some } (\text{remove-guards } x) = \text{Some } (\text{remove-guards } a) \oplus \text{Some } (\text{remove-guards } b)$
<proof>

lemma *no-guard-smaller:*
assumes $a \succeq b$
shows $\text{remove-guards } a \succeq \text{remove-guards } b$
<proof>

definition *add-empty-guards* :: $('i, 'a) \text{ heap} \Rightarrow ('i, 'a) \text{ heap}$ **where**
 $\text{add-empty-guards } h = (\text{get-fh } h, \text{Some } (\text{pwrite}, \{\#\}), (\lambda-. \text{Some } []))$

lemma *no-guard-map-empty-compatible:*
assumes $\text{no-guard } a$
and $\text{get-fh } b = \text{Map.empty}$
shows $a \#\# b$
<proof>

lemma *no-guard-add-empty-is-add:*
assumes $\text{no-guard } h$
shows $\text{Some } (\text{add-empty-guards } h) = \text{Some } h \oplus \text{Some } (\text{Map.empty}, \text{Some } (\text{pwrite}, \{\#\}), (\lambda-. \text{Some } []))$
<proof>

lemma *no-guard-and-sat-p-empty-guards:*
assumes $(s, h), (s', h') \models A$
and $\text{no-guard } h \wedge \text{no-guard } h'$
shows $(s, \text{add-empty-guards } h), (s', \text{add-empty-guards } h') \models \text{Star } A \text{ EmptyFull-Guards}$
<proof>

lemma *no-guard-add-empty-guards-sum:*
assumes $\text{no-guard } x$
and $\text{Some } x = \text{Some } a \oplus \text{Some } b$
shows $\text{Some } (\text{add-empty-guards } x) = \text{Some } (\text{add-empty-guards } a) \oplus \text{Some } b$
<proof>

lemma *semi-consistent-empty-no-guard-initial-value:*

assumes *no-guard* *h*
shows *semi-consistent* Γ (*view* Γ (*FractionalHeap.normalize* (*get-fh* *h*))) (*add-empty-guards* *h*)
 \langle *proof* \rangle

lemma *no-guards-remove-same*:
assumes *no-guard* *h*
shows $h = \text{remove-guards } (\text{add-empty-guards } h)$
 \langle *proof* \rangle

lemma *no-guards-remove*:
 $\text{no-guard } h \longleftrightarrow h = \text{remove-guards } h$
 \langle *proof* \rangle

definition *add-sguard-to-no-guard* :: ('i, 'a) heap \Rightarrow prat \Rightarrow 'a multiset \Rightarrow ('i, 'a) heap **where**
 $\text{add-sguard-to-no-guard } h \ \pi \ ms = (\text{get-fh } h, \text{Some } (\pi, ms), (\lambda-. \text{None}))$

lemma *get-fh-add-sguard*:
 $\text{get-fh } (\text{add-sguard-to-no-guard } h \ \pi \ ms) = \text{get-fh } h$
 \langle *proof* \rangle

lemma *add-sguard-as-sum*:
assumes *no-guard* *h*
shows $\text{Some } (\text{add-sguard-to-no-guard } h \ \pi \ ms) = \text{Some } h \oplus \text{Some } (\text{Map.empty}, \text{Some } (\pi, ms), (\lambda-. \text{None}))$
 \langle *proof* \rangle

definition *add-uguard-to-no-guard* :: 'i \Rightarrow ('i, 'a) heap \Rightarrow 'a list \Rightarrow ('i, 'a) heap **where**
 $\text{add-uguard-to-no-guard } k \ h \ l = (\text{get-fh } h, \text{None}, (\lambda-. \text{None}))(k := \text{Some } l)$

lemma *get-fh-add-uguard*:
 $\text{get-fh } (\text{add-uguard-to-no-guard } k \ h \ l) = \text{get-fh } h$
 \langle *proof* \rangle

lemma *prove-sum*:
assumes $a \ \#\# \ b$
and $\bigwedge x. \text{Some } x = \text{Some } a \oplus \text{Some } b \implies x = y$
shows $\text{Some } y = \text{Some } a \oplus \text{Some } b$
 \langle *proof* \rangle

lemma *add-uguard-as-sum*:
assumes *no-guard* *h*
shows $\text{Some } (\text{add-uguard-to-no-guard } k \ h \ l) = \text{Some } h \oplus \text{Some } (\text{Map.empty}, \text{None}, (\lambda-. \text{None}))(k := \text{Some } l)$
 \langle *proof* \rangle

lemma *no-guard-and-no-heap*:
assumes *Some h = Some p \oplus Some g*
and *no-guard p*
and *get-fh g = Map.empty*
shows *remove-guards h = p*
 \langle *proof* \rangle

lemma *decompose-guard-remove-easy*:
Some h = Some (remove-guards h) \oplus Some (Map.empty, get-gs h, get-gu h)
 \langle *proof* \rangle

lemma *all-guards-no-guard-propagates*:
assumes *all-guards x*
and *Some x = Some a \oplus Some b*
and *no-guard a*
shows *all-guards b*
 \langle *proof* \rangle

lemma *all-guards-exists-uargs*:
assumes *all-guards x*
shows \exists *uargs. \forall k. get-gu x k = Some (uargs k)*
 \langle *proof* \rangle

lemma *all-guards-sum-known-one*:
assumes *Some x = Some a \oplus Some b*
and *all-guards x*
and \bigwedge *k. get-gu a k = None*
and *get-gs a = Some (π , ms)*
shows \exists π' *msf uargs. (\forall k. get-gu b k = Some (uargs k)) \wedge*
 $((\pi = pwrite \wedge get-gs b = None \wedge msf = \{\#\}) \vee (pwrite = padd \pi \pi' \wedge get-gs
b = Some (π' , msf)))$
 \langle *proof* \rangle

fun *add-pwrite-option where*
add-pwrite-option None = None
 $|$ *add-pwrite-option (Some x) = Some (pwrite, x)*

definition *denormalize* :: *normal-heap \Rightarrow ('i, 'a) heap where*
denormalize H = ((λ l. add-pwrite-option (H l)), None, (λ -. None))

lemma *denormalize-properties*:
shows *no-guard (denormalize H)*
and *full-ownership (get-fh (denormalize H))*
and *normalize (get-fh (denormalize H)) = H*
and *full-ownership (get-fh h) \wedge no-guard h \implies denormalize (normalize (get-fh*

$h)) = h$
and *full-ownership* (*get-fh* h) \implies *denormalize* (*normalize* (*get-fh* h)) = *remove-guards* h
 \langle *proof* \rangle

lemma *no-guard-then-sat-star-uguard*:

assumes *no-guard* $h \wedge$ *no-guard* h'
and $(s, h), (s', h') \models Q$
shows $(s, \text{add-uguard-to-no-guard } k \ h \ (e \ s)), (s', \text{add-uguard-to-no-guard } k \ h' \ (e \ s')) \models \text{Star } Q \ (\text{UniqueGuard } k \ e)$
 \langle *proof* \rangle

lemma *no-guard-then-sat-star*:

assumes *no-guard* $h \wedge$ *no-guard* h'
and $(s, h), (s', h') \models Q$
shows $(s, \text{add-sguard-to-no-guard } h \ \pi \ (ms \ s)), (s', \text{add-sguard-to-no-guard } h' \ \pi \ (ms \ s')) \models \text{Star } Q \ (\text{SharedGuard } \pi \ ms)$
 \langle *proof* \rangle

end

4.3 Safety and Hoare Triples

In this file, the meaning of Hoare triples (Definition 4.1), through a notion of safety (see Section 4 and Appendix C). We also prove useful lemmas for the soundness proof.

theory *Safety*

imports *Guards*

begin

4.3.1 Preliminaries

definition *sat-inv* :: *store* \Rightarrow (i, a) *heap* \Rightarrow (i, a, nat) *single-context* \Rightarrow *bool*
where

sat-inv $s \ hj \ \Gamma \longleftrightarrow (s, hj), (s, hj) \models \text{invariant } \Gamma \wedge \text{no-guard } hj$

lemma *sat-invI*:

assumes $(s, hj), (s, hj) \models \text{invariant } \Gamma$

and *no-guard* hj

shows *sat-inv* $s \ hj \ \Gamma$

\langle *proof* \rangle

s and s' can differ on variables outside of *vars*, does not change anything.
upper-fvs $S \ \text{vars}$ means that *vars* is an upper-bound of "*fv* S "

definition *upper-fvs* :: $(\text{store} \times (i, a) \text{ heap}) \text{ set} \Rightarrow \text{var set} \Rightarrow \text{bool}$ **where**

upper-fvs $S \ \text{vars} \longleftrightarrow (\forall s \ s' \ h. (s, h) \in S \wedge \text{agrees vars } s \ s' \longrightarrow (s', h) \in S)$

Only need to agree on vars

definition *upperize where*

$upperize\ S\ vars = \{ \sigma' \mid \sigma\ \sigma'.\ \sigma \in S \wedge snd\ \sigma = snd\ \sigma' \wedge agrees\ vars\ (fst\ \sigma)\ (fst\ \sigma') \}$

definition *close-var where*

$close-var\ S\ x = \{ ((fst\ \sigma)(x := v),\ snd\ \sigma) \mid \sigma\ v.\ \sigma \in S \}$

lemma *upper-fvsI:*

assumes $\bigwedge s\ s'\ h.\ (s, h) \in S \wedge agrees\ vars\ s\ s' \implies (s', h) \in S$

shows $upper-fvs\ S\ vars$

<proof>

lemma *pair-sat-comm:*

assumes $pair-sat\ S\ S'\ A$

shows $pair-sat\ S'\ S\ A$

<proof>

lemma *in-upperize:*

$(s', h) \in upperize\ S\ vars \longleftrightarrow (\exists s.\ (s, h) \in S \wedge agrees\ vars\ s\ s')\ (\mathbf{is}\ ?A \longleftrightarrow ?B)$

<proof>

lemma *upper-fvs-upperize:*

$upper-fvs\ (upperize\ S\ vars)\ vars$

<proof>

lemma *upperize-larger:*

$S \subseteq upperize\ S\ vars$

<proof>

lemma *pair-sat-upperize:*

assumes $pair-sat\ S\ S'\ A$

shows $pair-sat\ (upperize\ S\ (fvA\ A))\ S'\ A$

<proof>

lemma *in-close-var:*

$(s', h) \in close-var\ S\ x \longleftrightarrow (\exists s\ v.\ (s, h) \in S \wedge s' = s(x := v))\ (\mathbf{is}\ ?A \longleftrightarrow ?B)$

<proof>

lemma *pair-sat-close-var:*

assumes $x \notin fvA\ A$

and $pair-sat\ S\ S'\ A$

shows $pair-sat\ (close-var\ S\ x)\ S'\ A$

<proof>

lemma *pair-sat-close-var-double:*

assumes $pair-sat\ S\ S'\ A$

and $x \notin fvA\ A$

shows $pair-sat\ (close-var\ S\ x)\ (close-var\ S'\ x)\ A$

$\langle \text{proof} \rangle$

lemma *close-var-subset*:

$S \subseteq \text{close-var } S \ x$

$\langle \text{proof} \rangle$

lemma *upper-fvs-close-vars*:

$\text{upper-fvs } (\text{close-var } S \ x) \ (- \ \{x\})$

$\langle \text{proof} \rangle$

lemma *sat-inv-agrees*:

assumes $\text{sat-inv } s \ hj \ \Gamma$

and $\text{agrees } (\text{fvA } (\text{invariant } \Gamma)) \ s \ s'$

shows $\text{sat-inv } s' \ hj \ \Gamma$

$\langle \text{proof} \rangle$

lemma *abort-iff-fvC*:

assumes $\text{agrees } (\text{fvC } C) \ s \ s'$

shows $\text{aborts } C \ (s, h) \longleftrightarrow \text{aborts } C \ (s', h)$

$\langle \text{proof} \rangle$

lemma *view-function-of-invE*:

assumes $\text{view-function-of-inv } \Gamma$

and $\text{sat-inv } s \ h \ \Gamma$

and $(h' :: ('i, 'a) \text{ heap}) \succeq h$

shows $\text{view } \Gamma \ (\text{normalize } (\text{get-fh } h)) = \text{view } \Gamma \ (\text{normalize } (\text{get-fh } h'))$

$\langle \text{proof} \rangle$

4.3.2 Safety

fun *no-abort* :: $('i, 'a, \text{nat}) \text{ cont} \Rightarrow \text{cmd} \Rightarrow \text{store} \Rightarrow ('i, 'a) \text{ heap} \Rightarrow \text{bool}$ **where**
 $\text{no-abort } \text{None } C \ s \ h \longleftrightarrow (\forall hf \ H. \text{Some } H = \text{Some } h \oplus \text{Some } hf \wedge \text{full-ownership}$
 $(\text{get-fh } H) \wedge \text{no-guard } H$

$\longrightarrow \neg \text{aborts } C \ (s, \text{normalize } (\text{get-fh } H))$

| $\text{no-abort } (\text{Some } \Gamma) \ C \ s \ h \longleftrightarrow (\forall hf \ H \ hj \ v0. \text{Some } H = \text{Some } h \oplus \text{Some } hj \oplus$
 $\text{Some } hf \wedge \text{full-ownership } (\text{get-fh } H) \wedge$

$\text{semi-consistent } \Gamma \ v0 \ H \wedge \text{sat-inv } s \ hj \ \Gamma$

$\longrightarrow \neg \text{aborts } C \ (s, \text{normalize } (\text{get-fh } H))$

lemma *no-abortI*:

assumes $\bigwedge (hf :: ('i, 'a) \text{ heap}) \ (H :: ('i, 'a) \text{ heap}). \text{Some } H = \text{Some } h \oplus \text{Some}$
 $hf \wedge \Delta = \text{None} \wedge \text{full-ownership } (\text{get-fh } H) \wedge \text{no-guard } H \implies \neg \text{aborts } C \ (s,$
 $\text{normalize } (\text{get-fh } H))$

and $\bigwedge H \ hf \ hj \ v0 \ \Gamma. \Delta = \text{Some } \Gamma \wedge \text{Some } H = \text{Some } h \oplus \text{Some } hj \oplus \text{Some}$
 $hf \wedge \text{full-ownership } (\text{get-fh } H) \wedge \text{semi-consistent } \Gamma \ v0 \ H \wedge \text{sat-inv } s \ hj \ \Gamma$

$\implies \neg \text{aborts } C \ (s, \text{normalize } (\text{get-fh } H))$

shows $\text{no-abort } \Delta \ C \ s \ (h :: ('i, 'a) \text{ heap})$

$\langle \text{proof} \rangle$

lemma *no-abortSomeI*:

assumes $\bigwedge H \text{ hf } hj \ v0. \text{ Some } H = \text{Some } h \oplus \text{Some } hj \oplus \text{Some } hf \wedge \text{full-ownership } (get\text{-fh } H) \wedge \text{semi-consistent } \Gamma \ v0 \ H \wedge \text{sat-inv } s \ hj \ \Gamma$
 $\implies \neg \text{aborts } C \ (s, \text{normalize } (get\text{-fh } H))$
shows *no-abort* $(\text{Some } \Gamma) \ C \ s \ (h :: ('i, 'a) \text{ heap})$
 $\langle \text{proof} \rangle$

lemma *no-abortNoneI*:

assumes $\bigwedge (hf :: ('i, 'a) \text{ heap}) \ (H :: ('i, 'a) \text{ heap}). \text{ Some } H = \text{Some } h \oplus \text{Some } hf \wedge \text{full-ownership } (get\text{-fh } H) \wedge \text{no-guard } H \implies \neg \text{aborts } C \ (s, \text{normalize } (get\text{-fh } H))$
shows *no-abort* $(\text{None} :: ('i, 'a, \text{nat}) \text{ cont}) \ C \ s \ (h :: ('i, 'a) \text{ heap})$
 $\langle \text{proof} \rangle$

lemma *no-abortE*:

assumes *no-abort* $\Delta \ C \ s \ h$
shows $\text{Some } H = \text{Some } h \oplus \text{Some } hf \implies \Delta = \text{None} \implies \text{full-ownership } (get\text{-fh } H) \implies \text{no-guard } H \implies \neg \text{aborts } C \ (s, \text{normalize } (get\text{-fh } H))$
and $\Delta = \text{Some } \Gamma \implies \text{Some } H = \text{Some } h \oplus \text{Some } hj \oplus \text{Some } hf \implies \text{sat-inv } s \ hj \ \Gamma \implies \text{full-ownership } (get\text{-fh } H) \implies \text{semi-consistent } \Gamma \ v0 \ H$
 $\implies \neg \text{aborts } C \ (s, \text{normalize } (get\text{-fh } H))$
 $\langle \text{proof} \rangle$

We define the notion of safety, central to the meaning of Hoare triples, as follows (Definition C.1 in the appendix).

fun *safe* :: $\text{nat} \Rightarrow ('i, 'a, \text{nat}) \text{ cont} \Rightarrow \text{cmd} \Rightarrow (\text{store} \times ('i, 'a) \text{ heap}) \Rightarrow (\text{store} \times ('i, 'a) \text{ heap}) \text{ set} \Rightarrow \text{bool}$ **where**
safe 0 - - - $\longleftrightarrow \text{True}$

$| \text{safe } (\text{Suc } n) \ \text{None} \ C \ (s, h) \ S \longleftrightarrow (C = \text{Cskip} \longrightarrow (s, h) \in S) \wedge \text{no-abort } (\text{None} :: ('i, 'a, \text{nat}) \text{ cont}) \ C \ s \ h \wedge \text{accesses } C \ s \subseteq \text{dom } (fst \ h) \wedge \text{writes } C \ s \subseteq \text{fpdom } (fst \ h) \wedge$
 $(\forall H \ \text{hf} \ C' \ s' \ h'. \text{ Some } H = \text{Some } h \oplus \text{Some } hf \wedge \text{full-ownership } (get\text{-fh } H) \wedge \text{no-guard } H$
 $\wedge \text{red } C \ (s, \text{normalize } (get\text{-fh } H)) \ C' \ (s', h')$
 $\longrightarrow (\exists h'' \ H'. \text{full-ownership } (get\text{-fh } H') \wedge \text{no-guard } H' \wedge h' = \text{normalize } (get\text{-fh } H') \wedge \text{Some } H' = \text{Some } h'' \oplus \text{Some } hf \wedge \text{safe } n \ (\text{None} :: ('i, 'a, \text{nat}) \text{ cont}) \ C' \ (s', h'') \ S))$

$| \text{safe } (\text{Suc } n) \ (\text{Some } \Gamma) \ C \ (s, h) \ S \longleftrightarrow (C = \text{Cskip} \longrightarrow (s, h) \in S) \wedge \text{no-abort } (\text{Some } \Gamma) \ C \ s \ h \wedge \text{accesses } C \ s \subseteq \text{dom } (fst \ h) \wedge \text{writes } C \ s \subseteq \text{fpdom } (fst \ h) \wedge$
 $(\forall H \ \text{hf} \ C' \ s' \ h' \ hj \ v0. \text{ Some } H = \text{Some } h \oplus \text{Some } hj \oplus \text{Some } hf \wedge \text{full-ownership } (get\text{-fh } H) \wedge \text{semi-consistent } \Gamma \ v0 \ H \wedge \text{sat-inv } s \ hj \ \Gamma$
 $\wedge \text{red } C \ (s, \text{normalize } (get\text{-fh } H)) \ C' \ (s', h')$
 $\longrightarrow (\exists h'' \ H' \ hj'. \text{full-ownership } (get\text{-fh } H') \wedge \text{semi-consistent } \Gamma \ v0 \ H' \wedge \text{sat-inv } s' \ hj' \ \Gamma$
 $\wedge h' = \text{normalize } (get\text{-fh } H') \wedge \text{Some } H' = \text{Some } h'' \oplus \text{Some } hj' \oplus \text{Some } hf \wedge \text{safe } n \ (\text{Some } \Gamma) \ C' \ (s', h'') \ S))$

lemma *safeNoneI*:

assumes $C = Cskip \implies (s, h) \in S$
and *no-abort* $None\ C\ s\ h$
and *accesses* $C\ s \subseteq dom\ (fst\ h) \wedge writes\ C\ s \subseteq fpdom\ (fst\ h)$
and $\bigwedge H\ hf\ C'\ s'\ h'. Some\ H = Some\ h \oplus Some\ hf \wedge full\text{-}ownership\ (get\text{-}fh\ H) \wedge no\text{-}guard\ H \wedge red\ C\ (s, normalize\ (get\text{-}fh\ H))\ C'\ (s', h')$
 $\implies (\exists h''\ H'. full\text{-}ownership\ (get\text{-}fh\ H') \wedge no\text{-}guard\ H' \wedge h' = normalize\ (get\text{-}fh\ H') \wedge Some\ H' = Some\ h'' \oplus Some\ hf \wedge safe\ n\ (None :: ('i, 'a, nat)\ cont)\ C'\ (s', h'')\ S)$
shows *safe* $(Suc\ n)\ (None :: ('i, 'a, nat)\ cont)\ C\ (s, h :: ('i, 'a)\ heap)\ S$
<proof>

lemma *safeSomeI*:

assumes $C = Cskip \implies (s, h) \in S$
and *no-abort* $(Some\ \Gamma)\ C\ s\ h$
and *accesses* $C\ s \subseteq dom\ (fst\ h) \wedge writes\ C\ s \subseteq fpdom\ (fst\ h)$
and $\bigwedge H\ hf\ C'\ s'\ h'\ hj\ v0. Some\ H = Some\ h \oplus Some\ hj \oplus Some\ hf \wedge full\text{-}ownership\ (get\text{-}fh\ H) \wedge semi\text{-}consistent\ \Gamma\ v0\ H \wedge sat\text{-}inv\ s\ hj\ \Gamma \wedge red\ C\ (s, normalize\ (get\text{-}fh\ H))\ C'\ (s', h')$
 $\implies (\exists h''\ H'\ hj'. full\text{-}ownership\ (get\text{-}fh\ H') \wedge semi\text{-}consistent\ \Gamma\ v0\ H' \wedge sat\text{-}inv\ s'\ hj'\ \Gamma \wedge h' = normalize\ (get\text{-}fh\ H') \wedge Some\ H' = Some\ h'' \oplus Some\ hj' \oplus Some\ hf \wedge safe\ n\ (Some\ \Gamma)\ C'\ (s', h'')\ S)$
shows *safe* $(Suc\ n)\ (Some\ \Gamma)\ C\ (s, h :: ('i, 'a)\ heap)\ S$
<proof>

lemma *safeI*:

fixes $\Delta :: ('i, 'a, nat)\ cont$
assumes $C = Cskip \implies (s, h) \in S$
and *no-abort* $\Delta\ C\ s\ h$
and *accesses* $C\ s \subseteq dom\ (fst\ h) \wedge writes\ C\ s \subseteq fpdom\ (fst\ h)$
and $\bigwedge H\ hf\ C'\ s'\ h'. \Delta = None \implies Some\ H = Some\ h \oplus Some\ hf \wedge full\text{-}ownership\ (get\text{-}fh\ H) \wedge no\text{-}guard\ H \wedge red\ C\ (s, normalize\ (get\text{-}fh\ H))\ C'\ (s', h')$
 $\implies (\exists h''\ H'. full\text{-}ownership\ (get\text{-}fh\ H') \wedge no\text{-}guard\ H' \wedge h' = normalize\ (get\text{-}fh\ H') \wedge Some\ H' = Some\ h'' \oplus Some\ hf \wedge safe\ n\ (None :: ('i, 'a, nat)\ cont)\ C'\ (s', h'')\ S)$
and $\bigwedge H\ hf\ C'\ s'\ h'\ hj\ v0\ \Gamma. \Delta = Some\ \Gamma \implies Some\ H = Some\ h \oplus Some\ hj \oplus Some\ hf \wedge full\text{-}ownership\ (get\text{-}fh\ H) \wedge semi\text{-}consistent\ \Gamma\ v0\ H \wedge sat\text{-}inv\ s\ hj\ \Gamma \wedge red\ C\ (s, normalize\ (get\text{-}fh\ H))\ C'\ (s', h')$
 $\implies (\exists h''\ H'\ hj'. full\text{-}ownership\ (get\text{-}fh\ H') \wedge semi\text{-}consistent\ \Gamma\ v0\ H' \wedge sat\text{-}inv\ s'\ hj'\ \Gamma \wedge h' = normalize\ (get\text{-}fh\ H') \wedge Some\ H' = Some\ h'' \oplus Some\ hj' \oplus Some\ hf \wedge safe\ n\ (Some\ \Gamma)\ C'\ (s', h'')\ S)$
shows *safe* $(Suc\ n)\ \Delta\ C\ (s, h :: ('i, 'a)\ heap)\ S$
<proof>

lemma *safeSomeAltI*:

assumes $C = Cskip \implies (s, h) \in S$
and $\bigwedge H hf hj v0. Some H = Some h \oplus Some hj \oplus Some hf \wedge full\text{-}ownership$
 $(get\text{-}fh H) \wedge semi\text{-}consistent \Gamma v0 H \wedge sat\text{-}inv s hj \Gamma$
 $\implies \neg aborts C (s, normalize (get\text{-}fh H))$
and $\bigwedge H hf C' s' h' hj v0. Some H = Some h \oplus Some hj \oplus Some hf \wedge$
 $full\text{-}ownership (get\text{-}fh H)$
 $\wedge semi\text{-}consistent \Gamma v0 H \wedge sat\text{-}inv s hj \Gamma \implies red C (s, normalize (get\text{-}fh$
 $H)) C' (s', h')$
 $\implies (\exists h'' H' hj'. full\text{-}ownership (get\text{-}fh H') \wedge semi\text{-}consistent \Gamma v0 H' \wedge sat\text{-}inv$
 $s' hj' \Gamma$
 $\wedge h' = normalize (get\text{-}fh H') \wedge Some H' = Some h'' \oplus Some hj' \oplus Some hf \wedge$
 $safe n (Some \Gamma) C' (s', h'') S)$
and $accesses C s \subseteq dom (fst h) \wedge writes C s \subseteq fpdom (fst h)$
shows $safe (Suc n) (Some \Gamma) C (s, h :: ('i, 'a) heap) S$
 $\langle proof \rangle$

lemma *safeAccessesE*:

assumes $safe (Suc n) \Delta C \sigma S$
shows $accesses C (fst \sigma) \subseteq dom (fst (snd \sigma)) \wedge writes C (fst \sigma) \subseteq fpdom (fst$
 $(snd \sigma))$
 $\langle proof \rangle$

lemma *safeSomeE*:

assumes $safe (Suc n) (Some \Gamma) C (s, h :: ('i, 'a) heap) S$
shows $C = Cskip \implies (s, h) \in S$
and $no\text{-}abort (Some \Gamma) C s h$
and $Some H = Some h \oplus Some hj \oplus Some hf \implies full\text{-}ownership (get\text{-}fh H)$
 $\implies semi\text{-}consistent \Gamma v0 H \implies sat\text{-}inv s hj \Gamma \implies red C (s, normalize$
 $(get\text{-}fh H)) C' (s', h')$
 $\implies (\exists h'' H' hj'. full\text{-}ownership (get\text{-}fh H') \wedge semi\text{-}consistent \Gamma v0 H' \wedge sat\text{-}inv$
 $s' hj' \Gamma$
 $\wedge h' = normalize (get\text{-}fh H') \wedge Some H' = Some h'' \oplus Some hj' \oplus Some hf \wedge$
 $safe n (Some \Gamma) C' (s', h'') S)$
 $\langle proof \rangle$

lemma *safeNoneE*:

assumes $safe (Suc n) (None :: ('i, 'a, nat) cont) C (s, h :: ('i, 'a) heap) S$
shows $C = Cskip \implies (s, h) \in S$
and $no\text{-}abort (None :: ('i, 'a, nat) cont) C s h$
and $Some H = Some h \oplus Some hf \implies full\text{-}ownership (get\text{-}fh H) \implies no\text{-}guard$
 $H \implies red C (s, normalize (get\text{-}fh H)) C' (s', h')$
 $\implies (\exists h'' H'. full\text{-}ownership (get\text{-}fh H') \wedge no\text{-}guard H' \wedge h' = normalize (get\text{-}fh$
 $H') \wedge Some H' = Some h'' \oplus Some hf \wedge safe n (None :: ('i, 'a, nat) cont) C' (s',$
 $h'') S)$
 $\langle proof \rangle$

lemma *safeNoneE-bis*:

fixes $no\text{-}cont :: ('i, 'a, nat) cont$
assumes $safe (Suc n) no\text{-}cont C (s, h :: ('i, 'a) heap) S$
and $no\text{-}cont = None$
shows $C = Cskip \implies (s, h) \in S$
and $no\text{-}abort no\text{-}cont C s h$
and $Some H = Some h \oplus Some hf \implies full\text{-}ownership (get\text{-}fh H) \implies no\text{-}guard$
 $H \implies red C (s, normalize (get\text{-}fh H)) C' (s', h')$
 $\implies (\exists h'' H'. full\text{-}ownership (get\text{-}fh H') \wedge no\text{-}guard H' \wedge h' = normalize (get\text{-}fh$
 $H') \wedge Some H' = Some h'' \oplus Some hf \wedge safe n no\text{-}cont C' (s', h'') S)$
 $\langle proof \rangle$

4.3.3 Useful results about safety

lemma $no\text{-}abort\text{-}larger$:

assumes $h' \succeq h$
and $no\text{-}abort \Gamma C s h$
shows $no\text{-}abort \Gamma C s h'$
 $\langle proof \rangle$

lemma $safe\text{-}larger\text{-}set\text{-}aux$:

fixes $\Delta :: ('i, 'a, nat) cont$
assumes $safe n \Delta C (s, h) S$
and $S \subseteq S'$
shows $safe n \Delta C (s, h) S'$
 $\langle proof \rangle$

lemma $safe\text{-}larger\text{-}set$:

assumes $safe n \Delta C \sigma S$
and $S \subseteq S'$
shows $safe n \Delta C \sigma S'$
 $\langle proof \rangle$

lemma $safe\text{-}smaller\text{-}aux$:

fixes $\Delta :: ('i, 'a, nat) cont$
assumes $m \leq n$
and $safe n \Delta C (s, h) S$
shows $safe m \Delta C (s, h) S$
 $\langle proof \rangle$

lemma $safe\text{-}smaller$:

assumes $m \leq n$
and $safe n \Delta C \sigma S$
shows $safe m \Delta C \sigma S$
 $\langle proof \rangle$

If it is safe to execute n steps of C in the state (s_0, h) , then it is also safe to execute it in the state (s_1, h) , provided that s_0 and s_1 agree on the values of variables that are free in C , the invariant, and the postcondition.

lemma $safe\text{-}free\text{-}vars\text{-}aux$:

fixes $\Delta :: ('i, 'a, nat) cont$
assumes $safe\ n\ \Delta\ C\ (s0, h)\ S$
and $agrees\ (fvC\ C\ \cup\ vars)\ s0\ s1$
and $upper\text{-}fvs\ S\ vars$
and $\bigwedge \Gamma. \Delta = Some\ \Gamma \implies agrees\ (fvA\ (invariant\ \Gamma))\ s0\ s1$
shows $safe\ n\ \Delta\ C\ (s1, h)\ S$
 $\langle proof \rangle$

lemma *safe-free-vars-None*:
assumes $safe\ n\ (None :: ('i, 'a, nat) cont)\ C\ (s, h)\ S$
and $agrees\ (fvC\ C\ \cup\ vars)\ s\ s'$
and $upper\text{-}fvs\ S\ vars$
shows $safe\ n\ (None :: ('i, 'a, nat) cont)\ C\ (s', h)\ S$
 $\langle proof \rangle$

lemma *safe-free-vars-Some*:
assumes $safe\ n\ (Some\ \Gamma)\ C\ (s, h)\ S$
and $agrees\ (fvC\ C\ \cup\ vars\ \cup\ fvA\ (invariant\ \Gamma))\ s\ s'$
and $upper\text{-}fvs\ S\ vars$
shows $safe\ n\ (Some\ \Gamma)\ C\ (s', h)\ S$
 $\langle proof \rangle$

lemma *safe-free-vars*:
fixes $\Delta :: ('i, 'a, nat) cont$
assumes $safe\ n\ \Delta\ C\ (s, h)\ S$
and $agrees\ (fvC\ C\ \cup\ vars)\ s\ s'$
and $upper\text{-}fvs\ S\ vars$
and $\bigwedge \Gamma. \Delta = Some\ \Gamma \implies agrees\ (fvA\ (invariant\ \Gamma))\ s\ s'$
shows $safe\ n\ \Delta\ C\ (s', h)\ S$
 $\langle proof \rangle$

lemma *restrict-safe-to-bounded*:
assumes $safe\ n\ \Delta\ C\ (s, h)\ S$
and $bounded\ h$
shows $safe\ n\ \Delta\ C\ (s, h)\ (Set.filter\ (bounded\ \circ\ snd)\ S)$
 $\langle proof \rangle$

4.3.4 Hoare triples

The following defines when Hoare triples are valid, based on Definition 4.1.

definition *hoare-triple-valid* $:: ('i, 'a, nat) cont \Rightarrow ('i, 'a, nat) assertion \Rightarrow cmd$
 $\Rightarrow ('i, 'a, nat) assertion \Rightarrow bool$
 $(\langle - \models \{-\} - \{-\} \rangle [51, 0, 0] 81) \text{ where}$
 $hoare\text{-}triple\text{-}valid\ \Gamma\ P\ C\ Q \iff (\exists \Sigma. (\forall \sigma\ n. \sigma, \sigma \models P \wedge bounded\ (snd\ \sigma) \longrightarrow$
 $safe\ n\ \Gamma\ C\ \sigma\ (\Sigma\ \sigma)) \wedge$
 $(\forall \sigma\ \sigma'. \sigma, \sigma' \models P \longrightarrow pair\text{-}sat\ (\Sigma\ \sigma)\ (\Sigma\ \sigma')\ Q))$

lemma *hoare-triple-validI*:

assumes $\bigwedge s h n. (s, h), (s, h) \models P \implies \text{safe } n \Gamma C (s, h) (\Sigma (s, h))$
and $\bigwedge s h s' h'. (s, h), (s', h') \models P \implies \text{pair-sat } (\Sigma (s, h)) (\Sigma (s', h')) Q$
shows *hoare-triple-valid* $\Gamma P C Q$
<proof>

lemma *hoare-triple-validI-bounded*:

assumes $\bigwedge s h n. (s, h), (s, h) \models P \implies \text{bounded } h \implies \text{safe } n \Gamma C (s, h) (\Sigma (s, h))$
and $\bigwedge s h s' h'. (s, h), (s', h') \models P \implies \text{pair-sat } (\Sigma (s, h)) (\Sigma (s', h')) Q$
shows *hoare-triple-valid* $\Gamma P C Q$
<proof>

lemma *hoare-triple-valid-smallerI*:

assumes $\bigwedge \sigma n. \sigma, \sigma \models P \implies \text{safe } n \Gamma C \sigma (\Sigma \sigma)$
and $\bigwedge \sigma \sigma'. \sigma, \sigma' \models P \implies \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') Q$
shows *hoare-triple-valid* $\Gamma P C Q$
<proof>

lemma *hoare-triple-valid-smallerI-bounded*:

assumes $\bigwedge \sigma n. \sigma, \sigma \models P \implies \text{bounded } (\text{snd } \sigma) \implies \text{safe } n \Gamma C \sigma (\Sigma \sigma)$
and $\bigwedge \sigma \sigma'. \sigma, \sigma' \models P \implies \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') Q$
shows *hoare-triple-valid* $\Gamma P C Q$
<proof>

lemma *hoare-triple-validE*:

assumes *hoare-triple-valid* $\Gamma P C Q$
shows $\exists \Sigma. (\forall \sigma n. \sigma, \sigma \models P \wedge \text{bounded } (\text{snd } \sigma) \longrightarrow \text{safe } n \Gamma C \sigma (\Sigma \sigma)) \wedge$
 $(\forall \sigma \sigma'. \sigma, \sigma' \models P \longrightarrow \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') Q)$
<proof>

lemma *hoare-triple-valid-simplerE*:

assumes *hoare-triple-valid* $\Gamma P C Q$
and $\sigma, \sigma' \models P$
and *bounded* $(\text{snd } \sigma)$
and *bounded* $(\text{snd } \sigma')$
shows $\exists S S'. \text{safe } n \Gamma C \sigma S \wedge \text{safe } n \Gamma C \sigma' S' \wedge \text{pair-sat } S S' Q$
<proof>

end

4.4 Soundness of the Rules

In this file, we prove that each rule of the logic is sound. We do this by assuming that the Hoare triples in the premise of the rule hold semantically (as defined in `Safety.thy`), and then proving that the Hoare triple in the conclusion also holds semantically. We prove soundness of the logic (with some corollaries) at the end of the file.

For each rule, we first prove an important lemma about the safety of the statement (i.e., under which conditions is executing this statement safe, and what conditions will hold about the set of states that can be reached by executing this statement). We then use this lemma to prove the rule of the logic, by constructing the set of states that will be reached, proving that safety holds, and proving that the final set of states satisfies the postcondition.

```
theory Soundness
  imports Safety AbstractCommutativity
begin
```

4.4.1 Skip

```
lemma safe-skip:
  fixes  $\Delta :: ('i, 'a, nat) cont$ 
  assumes  $(s, h) \in S$ 
  shows  $safe\ n\ \Delta\ Cskip\ (s, h)\ S$ 
   $\langle proof \rangle$ 
```

```
theorem rule-skip:
  hoare-triple-valid  $\Gamma\ P\ Cskip\ P$ 
   $\langle proof \rangle$ 
```

4.4.2 Assign

```
inductive-cases red-assign-cases:  $red\ (Cassign\ x\ E)\ \sigma\ C'\ \sigma'$ 
inductive-cases aborts-assign-cases:  $aborts\ (Cassign\ x\ E)\ \sigma$ 
```

```
lemma safe-assign:
  fixes  $\Delta :: ('i, 'a, nat) cont$ 
  assumes  $\bigwedge \Gamma. \Delta = Some\ \Gamma \implies x \notin fvA\ (invariant\ \Gamma)$ 
  shows  $safe\ m\ \Delta\ (Cassign\ x\ E)\ (s, h)\ \{ (s(x := edenot\ E\ s), h) \}$ 
   $\langle proof \rangle$ 
```

```
theorem assign-rule:
  fixes  $\Delta :: ('i, 'a, nat) cont$ 
  assumes  $\bigwedge \Gamma. \Delta = Some\ \Gamma \implies x \notin fvA\ (invariant\ \Gamma)$ 
  and collect-existentials  $P \cap fvE\ E = \{\}$ 
  shows hoare-triple-valid  $\Delta\ (subA\ x\ E\ P)\ (Cassign\ x\ E)\ P$ 
   $\langle proof \rangle$ 
```

4.4.3 Alloc

```
inductive-cases red-alloc-cases:  $red\ (Calloc\ x\ E)\ \sigma\ C'\ \sigma'$ 
inductive-cases aborts-alloc-cases:  $aborts\ (Calloc\ x\ E)\ \sigma$ 
```

```
lemma safe-new-None:
```

safe n (*None* :: ('i, 'a, nat) cont) (*Calloc x E*) (*s*, (*Map.empty*, *gs*, *gu*)) { (*s(x := a)*, (*Map.empty(a ↦ (pwrite, edenot E s))*), *gs*, *gu*) | *a. True* }
 ⟨*proof*⟩

lemma *safe-new-Some*:

assumes $x \notin \text{fvA}$ (*invariant* Γ)
and *view-function-of-inv* Γ
shows *safe n* (*Some* Γ) (*Calloc x E*) (*s*, (*Map.empty*, *gs*, *gu*)) { (*s(x := a)*, (*Map.empty(a ↦ (pwrite, edenot E s))*), *gs*, *gu*) | *a. True* }
 ⟨*proof*⟩

lemma *safe-new*:

fixes $\Delta :: ('i, 'a, \text{nat}) \text{ cont}$
assumes $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies x \notin \text{fvA}$ (*invariant* Γ) \wedge *view-function-of-inv* Γ
shows *safe n* Δ (*Calloc x E*) (*s*, (*Map.empty*, *gs*, *gu*)) { (*s(x := a)*, (*Map.empty(a ↦ (pwrite, edenot E s))*), *gs*, *gu*) | *a. True* }
 ⟨*proof*⟩

theorem *new-rule*:

fixes $\Delta :: ('i, 'a, \text{nat}) \text{ cont}$
assumes $x \notin \text{fvE } E$
and $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies x \notin \text{fvA}$ (*invariant* Γ) \wedge *view-function-of-inv* Γ
shows *hoare-triple-valid* Δ *Emp* (*Calloc x E*) (*PointsTo* (*Evar x*) *pwrite E*)
 ⟨*proof*⟩

4.4.4 Write

inductive-cases *red-write-cases*: *red* (*Cwrite x E*) σ $C' \sigma'$

inductive-cases *aborts-write-cases*: *aborts* (*Cwrite x E*) σ

lemma *safe-write-None*:

assumes *fh* (*edenot loc s*) = *Some* (*pwrite*, *v*)
shows *safe n* (*None* :: ('i, 'a, nat) cont) (*Cwrite loc E*) (*s*, (*fh*, *gs*, *gu*)) { (*s*, (*fh(edenot loc s ↦ (pwrite, edenot E s))*), *gs*, *gu*) }
 ⟨*proof*⟩

lemma *safe-write-Some*:

assumes *fh* (*edenot loc s*) = *Some* (*pwrite*, *v*)
and *view-function-of-inv* Γ
shows *safe n* (*Some* Γ) (*Cwrite loc E*) (*s*, (*fh*, *gs*, *gu*)) { (*s*, (*fh(edenot loc s ↦ (pwrite, edenot E s))*), *gs*, *gu*) }
 ⟨*proof*⟩

lemma *safe-write*:

fixes $\Delta :: ('i, 'a, nat) cont$
assumes $fh (edenot loc s) = Some (pwrite, v)$
and $\bigwedge \Gamma. \Delta = Some \Gamma \implies view\text{-function-of-inv } \Gamma$
shows $safe\ n\ \Delta\ (Cwrite\ loc\ E)\ (s, (fh, gs, gu))\ \{ (s, (fh(edenot\ loc\ s \mapsto (pwrite, edenot\ E\ s)), gs, gu))\}$
 $\langle proof \rangle$

theorem *write-rule*:

fixes $\Delta :: ('i, 'a, nat) cont$
assumes $\bigwedge \Gamma. \Delta = Some \Gamma \implies view\text{-function-of-inv } \Gamma$
and $v \notin fvE\ loc$
shows $hoare\text{-triple-valid } \Delta\ (Exists\ v\ (PointsTo\ loc\ pwrite\ (Evar\ v)))\ (Cwrite\ loc\ E)\ (PointsTo\ loc\ pwrite\ E)$
 $\langle proof \rangle$

4.4.5 Read

inductive-cases *red-read-cases*: $red\ (Cread\ x\ E)\ \sigma\ C'\ \sigma'$

inductive-cases *aborts-read-cases*: $aborts\ (Cread\ x\ E)\ \sigma$

lemma *safe-read-None*:

$safe\ n\ (None :: ('i, 'a, nat) cont)\ (Cread\ x\ E)\ (s, ([edenot\ E\ s \mapsto (\pi, v)], gs, gu))$
 $\{ (s(x := v), ([edenot\ E\ s \mapsto (\pi, v)], gs, gu))\}$
 $\langle proof \rangle$

lemma *safe-read-Some*:

assumes $view\text{-function-of-inv } \Gamma$
and $x \notin fvA\ (invariant\ \Gamma)$
shows $safe\ n\ (Some\ \Gamma)\ (Cread\ x\ E)\ (s, ([edenot\ E\ s \mapsto (\pi, v)], gs, gu))\ \{ (s(x := v), ([edenot\ E\ s \mapsto (\pi, v)], gs, gu))\}$
 $\langle proof \rangle$

lemma *safe-read*:

fixes $\Delta :: ('i, 'a, nat) cont$
assumes $\bigwedge \Gamma. \Delta = Some \Gamma \implies x \notin fvA\ (invariant\ \Gamma) \wedge view\text{-function-of-inv } \Gamma$
shows $safe\ n\ \Delta\ (Cread\ x\ E)\ (s, ([edenot\ E\ s \mapsto (\pi, v)], gs, gu))\ \{ (s(x := v), ([edenot\ E\ s \mapsto (\pi, v)], gs, gu))\}$
 $\langle proof \rangle$

theorem *read-rule*:

fixes $\Delta :: ('i, 'a, nat) cont$
assumes $\bigwedge \Gamma. \Delta = Some \Gamma \implies x \notin fvA\ (invariant\ \Gamma) \wedge view\text{-function-of-inv } \Gamma$
and $x \notin fvE\ E \cup fvE\ e$
shows $hoare\text{-triple-valid } \Delta\ (PointsTo\ E\ \pi\ e)\ (Cread\ x\ E)\ (And\ (PointsTo\ E\ \pi\ e)\ (Bool\ (Beq\ (Evar\ x)\ e)))$
 $\langle proof \rangle$

4.4.6 Share

lemma *share-no-abort*:

assumes *no-abort* (*Some* Γ) *C s* ($h :: ('i, 'a)$ *heap*)
and *Some* ($h' :: ('i, 'a)$ *heap*) = *Some* $h \oplus$ *Some* h_j
and *sat-inv* s h_j Γ
and *get-gs* h = *Some* (*pwrite*, *sargs*)
and $\bigwedge k.$ *get-gu* h k = *Some* (*uargs* k)
and *reachable-value* (*saction* Γ) (*uaction* Γ) v_0 *sargs* *uargs* (*view* Γ (*normalize*
(*get-fh* h_j)))
and *view-function-of-inv* Γ
shows *no-abort* *None* *C s* (*remove-guards* h')
 \langle *proof* \rangle

definition *S-after-share* **where**

S-after-share S Γ v_0 = $\{ (s, \text{remove-guards } h') \mid h_j h' s. \text{semi-consistent } \Gamma v_0$
 $h' \wedge \text{Some } h' = \text{Some } h \oplus \text{Some } h_j \wedge (s, h) \in S \wedge \text{sat-inv } s h_j \Gamma \}$

lemma *share-lemma*:

assumes *safe n* (*Some* Γ) *C* ($s, h :: ('i, 'a)$ *heap*) S
and *Some* ($h' :: ('i, 'a)$ *heap*) = *Some* $h \oplus$ *Some* h_j
and *sat-inv* s h_j Γ
and *semi-consistent* Γ v_0 h'
and *view-function-of-inv* Γ
and *bounded* h'
shows *safe n* (*None* :: ($'i, 'a, \text{nat}$) *cont*) *C* ($s, \text{remove-guards } h'$) (*S-after-share*
 S Γ v_0)
 \langle *proof* \rangle

definition *no-need-guards* **where**

no-need-guards $A \iff (\forall s_1 h_1 s_2 h_2. (s_1, h_1), (s_2, h_2) \models A \implies (s_1, \text{re-}$
 $\text{move-guards } h_1), (s_2, \text{remove-guards } h_2) \models A)$

lemma *has-guard-then-safe-none*:

assumes \neg *no-guard* h
and $C = \text{Cskip} \implies (s, h) \in S$
and *accesses* C $s \subseteq \text{dom}(\text{fst } h) \wedge \text{writes } C$ $s \subseteq \text{fpdom}(\text{fst } h)$
shows *safe n* (*None* :: ($'i, 'a, \text{nat}$) *cont*) *C* (s, h) S
 \langle *proof* \rangle

theorem *share-rule*:

fixes $\Gamma :: ('i, 'a, \text{nat})$ *single-context*
assumes $\Gamma = \langle \text{view} = f, \text{abstract-view} = \alpha, \text{saction} = \text{sact}, \text{uaction} = \text{uact},$
*invariant} = J \rangle
and *all-axioms* α *sact spre uact upre*
and *hoare-triple-valid* (*Some* Γ) (*Star* P *EmptyFullGuards*) *C* (*Star* Q (*And*
(*PreSharedGuards* (*Abs-precondition spre*)) (*PreUniqueGuards* (*Abs-indexed-precondition*
upre))))
and *view-function-of-inv* Γ*

and unary $J \wedge$ precise J
and wf-indexed-precondition upre \wedge wf-precondition spre
and $x \notin \text{fv}A J$
and no-guard-assertion (Star P (LowView $(\alpha \circ f)$ $J x$))
shows hoare-triple-valid (None :: ('i, 'a, nat) cont) (Star P (LowView $(\alpha \circ f)$ $J x$)) C (Star Q (LowView $(\alpha \circ f)$ $J x$))
 <proof>

4.4.7 Atomic

lemma red-rtrans-induct:

assumes red-rtrans $C \sigma C' \sigma'$
and $\bigwedge C \sigma. P C \sigma C \sigma$
and $\bigwedge C \sigma C' \sigma' C'' \sigma''. \text{red } C \sigma C' \sigma' \implies \text{red-rtrans } C' \sigma' C'' \sigma'' \implies P C' \sigma' C'' \sigma'' \implies P C \sigma C'' \sigma''$
shows $P C \sigma C' \sigma'$
 <proof>

lemma safe-atomic:

assumes red-rtrans $C1 \sigma1 C2 \sigma2$
and $\sigma1 = (s1, H1)$
and $\sigma2 = (s2, H2)$
and $\bigwedge n. \text{safe } n$ (None :: ('i, 'a, nat) cont) $C1 (s1, h) S$
and $H = \text{denormalize } H1$
and Some $H = \text{Some } h \oplus \text{Some } hf$
and full-ownership (get-fh H) \wedge no-guard H
shows $\neg \text{aborts } C2 \sigma2 \wedge (C2 = \text{Cskip} \longrightarrow (\exists h1 H'. \text{Some } H' = \text{Some } h1 \oplus \text{Some } hf \wedge H2 = \text{normalize } (\text{get-fh } (H'))) \wedge \text{no-guard } H' \wedge \text{full-ownership } (\text{get-fh } H') \wedge (s2, h1) \in S)$
 <proof>

theorem atomic-rule-unique:

fixes $\Gamma :: ('i, 'a, nat)$ single-context

fixes map-to-list :: nat \Rightarrow 'a list

fixes map-to-arg :: nat \Rightarrow 'a

assumes $\Gamma = (\mid \text{view} = f, \text{abstract-view} = \alpha, \text{saction} = \text{sact}, \text{uaction} = \text{uact}, \text{invariant} = J \mid)$

and hoare-triple-valid (None :: ('i, 'a, nat) cont) (Star P (View $f J (\lambda s. s x)$))
 C (Star Q (View $f J (\lambda s. \text{uact index } (s x) (\text{map-to-arg } (s \text{uarg}))))))$

and precise $J \wedge$ unary J

and view-function-of-inv Γ

and $x \notin \text{fv}C C \cup \text{fv}A P \cup \text{fv}A Q \cup \text{fv}A J$

and uarg $\notin \text{fv}C C$

and $l \notin \text{fv}C C$

and $x \notin \text{fv}S (\lambda s. \text{map-to-list } (s \ l))$
and $x \notin \text{fv}S (\lambda s. \text{map-to-arg } (s \ \text{uarg}) \ \# \ \text{map-to-list } (s \ l))$

and *no-guard-assertion* P
and *no-guard-assertion* Q

shows *hoare-triple-valid* (*Some* Γ) (*Star* P (*UniqueGuard* *index* ($\lambda s. \text{map-to-list } (s \ l)$))) (*Catomic* C)
(*Star* Q (*UniqueGuard* *index* ($\lambda s. \text{map-to-arg } (s \ \text{uarg}) \ \#$
map-to-list ($s \ l$))))
⟨*proof*⟩

theorem *atomic-rule-shared*:

fixes $\Gamma :: ('i, 'a, \text{nat}) \ \text{single-context}$

fixes *map-to-multiset* $:: \text{nat} \Rightarrow 'a \ \text{multiset}$
fixes *map-to-arg* $:: \text{nat} \Rightarrow 'a$

assumes $\Gamma = (\langle \text{view} = f, \text{abstract-view} = \alpha, \text{saction} = \text{sact}, \text{uaction} = \text{uact}, \text{invariant} = J \rangle)$
and *hoare-triple-valid* (*None* $:: ('i, 'a, \text{nat}) \ \text{cont}$) (*Star* P (*View* $f \ J$ ($\lambda s. \text{sact } (s \ x)$))) C
(*Star* Q (*View* $f \ J$ ($\lambda s. \text{sact } (s \ x) \ (\text{map-to-arg } (s \ \text{sarg}))))$))
and *precise* $J \wedge \text{unary } J$
and *view-function-of-inv* Γ
and $x \notin \text{fv}C \ C \cup \text{fv}A \ P \cup \text{fv}A \ Q \cup \text{fv}A \ J$

and $\text{sarg} \notin \text{fv}C \ C$
and $\text{ms} \notin \text{fv}C \ C$

and $x \notin \text{fv}S (\lambda s. \text{map-to-multiset } (s \ \text{ms}))$
and $x \notin \text{fv}S (\lambda s. \{\# \ \text{map-to-arg } (s \ \text{sarg}) \ \#\} + \text{map-to-multiset } (s \ \text{ms}))$

and *no-guard-assertion* P
and *no-guard-assertion* Q

shows *hoare-triple-valid* (*Some* Γ) (*Star* P (*SharedGuard* π ($\lambda s. \text{map-to-multiset } (s \ \text{ms})$))) (*Catomic* C)
(*Star* Q (*SharedGuard* π ($\lambda s. \{\# \ \text{map-to-arg } (s \ \text{sarg}) \ \#\} + \text{map-to-multiset } (s \ \text{ms})$)))
⟨*proof*⟩

4.4.8 Parallel

lemma *par-cases*:

assumes *red* (*Cpar* $C1 \ C2$) $\sigma \ C' \ \sigma'$
and $\bigwedge C1'. \ C' = \text{Cpar } C1' \ C2 \wedge \text{red } C1 \ \sigma \ C1' \ \sigma' \implies P$
and $\bigwedge C2'. \ C' = \text{Cpar } C1 \ C2' \wedge \text{red } C2 \ \sigma \ C2' \ \sigma' \implies P$

and $C1 = Cskip \wedge C2 = Cskip \wedge C' = Cskip \wedge \sigma = \sigma' \implies P$
shows P
 ⟨proof⟩

lemma *no-abort-par*:

assumes *no-abort* Γ $C1$ s h
and *no-abort* Γ $C2$ s h
and *safe* $(Suc\ n)$ Δ $C1$ $(s, h1)$ $S1$
and *safe* $(Suc\ n)$ Δ $C2$ $(s, h2)$ $S2$
and $Some\ h = Some\ h1 \oplus Some\ h2$
and *bounded* h
shows *no-abort* Γ $(Cpar\ C1\ C2)$ s $h \wedge accesses\ (Cpar\ C1\ C2)\ s \subseteq dom\ (fst\ h)$
 $\wedge writes\ (Cpar\ C1\ C2)\ s \subseteq fpdom\ (fst\ h)$
 ⟨proof⟩

lemma *parallel-comp-none*:

assumes *safe* n $(None :: ('i, 'a, nat)\ cont)$ $C1$ $(s, h1)$ $S1$
and *safe* n $(None :: ('i, 'a, nat)\ cont)$ $C2$ $(s, h2)$ $S2$
and $Some\ h = Some\ h1 \oplus Some\ h2$

and *disjoint* $(fvC\ C1 \cup vars1)$ $(wrC\ C2)$
and *disjoint* $(fvC\ C2 \cup vars2)$ $(wrC\ C1)$

and *upper-fvs* $S1$ $vars1$
and *upper-fvs* $S2$ $vars2$

and *bounded* h

shows *safe* n $(None :: ('i, 'a, nat)\ cont)$ $(Cpar\ C1\ C2)$ (s, h) $(add-states\ S1\ S2)$
 ⟨proof⟩

lemma *parallel-comp-some*:

assumes *safe* n $(Some\ \Gamma)$ $C1$ $(s, h1)$ $S1$
and *safe* n $(Some\ \Gamma)$ $C2$ $(s, h2)$ $S2$
and $Some\ h = Some\ h1 \oplus Some\ h2$

and *disjoint* $(fvC\ C1 \cup vars1)$ $(wrC\ C2)$
and *disjoint* $(fvC\ C2 \cup vars2)$ $(wrC\ C1)$

and *upper-fvs* $S1$ $vars1$
and *upper-fvs* $S2$ $vars2$

and *disjoint* $(fvA\ (invariant\ \Gamma))$ $(wrC\ C2)$
and *disjoint* $(fvA\ (invariant\ \Gamma))$ $(wrC\ C1)$

and *bounded* h

shows *safe n* (Some Γ) (Cpar C1 C2) (s, h) (add-states S1 S2)
 ⟨proof⟩

lemma *parallel-comp*:

fixes $\Delta :: ('i, 'a, nat)$ cont

assumes *safe n* Δ C1 (s, h1) S1

and *safe n* Δ C2 (s, h2) S2

and Some h = Some h1 \oplus Some h2

and disjoint (fvC C1 \cup vars1) (wrC C2)

and disjoint (fvC C2 \cup vars2) (wrC C1)

and upper-fvs S1 vars1

and upper-fvs S2 vars2

and $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies \text{disjoint (fvA (invariant } \Gamma)) (wrC C2)$

and $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies \text{disjoint (fvA (invariant } \Gamma)) (wrC C1)$

and bounded h

shows *safe n* Δ (Cpar C1 C2) (s, h) (add-states S1 S2)
 ⟨proof⟩

theorem *rule-par*:

fixes $\Delta :: ('i, 'a, nat)$ cont

assumes *hoare-triple-valid* Δ P1 C1 Q1

and *hoare-triple-valid* Δ P2 C2 Q2

and disjoint (fvA P1 \cup fvC C1 \cup fvA Q1) (wrC C2)

and disjoint (fvA P2 \cup fvC C2 \cup fvA Q2) (wrC C1)

and $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies \text{disjoint (fvA (invariant } \Gamma)) (wrC C2)$

and $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies \text{disjoint (fvA (invariant } \Gamma)) (wrC C1)$

and precise P1 \vee precise P2

shows *hoare-triple-valid* Δ (Star P1 P2) (Cpar C1 C2) (Star Q1 Q2)
 ⟨proof⟩

4.4.9 If

lemma *if-cases*:

assumes *red* (Cif b C1 C2) (s, h) C' (s', h')

and C' = C1 $\implies s = s' \wedge h = h' \implies \text{bdenot } b \ s \implies P$

and C' = C2 $\implies s = s' \wedge h = h' \implies \neg \text{bdenot } b \ s \implies P$

shows P

⟨proof⟩

lemma *if-safe-None*:

fixes $\Delta :: ('i, 'a, nat) cont$

assumes $bdenot\ b\ s \implies safe\ n\ \Delta\ C1\ (s, h)\ S$

and $\neg\ bdenot\ b\ s \implies safe\ n\ \Delta\ C2\ (s, h)\ S$

and $\Delta = None$

shows $safe\ (Suc\ n)\ (None :: ('i, 'a, nat) cont)\ (Cif\ b\ C1\ C2)\ (s, h)\ S$

<proof>

lemma *if-safe-Some*:

assumes $bdenot\ b\ s \implies safe\ n\ (Some\ \Gamma)\ C1\ (s, h)\ S$

and $\neg\ bdenot\ b\ s \implies safe\ n\ (Some\ \Gamma)\ C2\ (s, h)\ S$

shows $safe\ (Suc\ n)\ (Some\ \Gamma)\ (Cif\ b\ C1\ C2)\ (s, h)\ S$

<proof>

lemma *if-safe*:

fixes $\Delta :: ('i, 'a, nat) cont$

assumes $bdenot\ b\ s \implies safe\ n\ \Delta\ C1\ (s, h)\ S$

and $\neg\ bdenot\ b\ s \implies safe\ n\ \Delta\ C2\ (s, h)\ S$

shows $safe\ (Suc\ n)\ \Delta\ (Cif\ b\ C1\ C2)\ (s, h)\ S$

<proof>

theorem *if1-rule*:

fixes $\Delta :: ('i, 'a, nat) cont$

assumes $hoare\ triple\ valid\ \Delta\ (And\ P\ (Bool\ b))\ C1\ Q$

and $hoare\ triple\ valid\ \Delta\ (And\ P\ (Bool\ (Bnot\ b)))\ C2\ Q$

shows $hoare\ triple\ valid\ \Delta\ (And\ P\ (Low\ b))\ (Cif\ b\ C1\ C2)\ Q$

<proof>

theorem *if2-rule*:

fixes $\Delta :: ('i, 'a, nat) cont$

assumes $hoare\ triple\ valid\ \Delta\ (And\ P\ (Bool\ b))\ C1\ Q$

and $hoare\ triple\ valid\ \Delta\ (And\ P\ (Bool\ (Bnot\ b)))\ C2\ Q$

and *unary* Q

shows $hoare\ triple\ valid\ \Delta\ P\ (Cif\ b\ C1\ C2)\ Q$

<proof>

4.4.10 Sequential composition

inductive-cases *red-seq-cases*: $red\ (Cseq\ C1\ C2)\ \sigma\ C'\ \sigma'$

lemma *aborts-seq-aborts-C1*:

assumes $aborts\ (Cseq\ C1\ C2)\ \sigma$

shows $aborts\ C1\ \sigma$

<proof>

lemma *safe-seq-None*:

assumes $\text{safe } n \text{ (None :: ('i, 'a, nat) cont) } C1 \text{ (s, h) } S1$
and $\bigwedge m \ s' \ h'. m \leq n \wedge (s', h') \in S1 \implies \text{safe } m \text{ (None :: ('i, 'a, nat) cont) } C2 \text{ (s', h') } S2$
shows $\text{safe } n \text{ (None :: ('i, 'a, nat) cont) (Cseq } C1 \ C2) \text{ (s, h) } S2$
 $\langle \text{proof} \rangle$

lemma *safe-seq-Some*:

assumes $\text{safe } n \text{ (Some } \Gamma) \ C1 \text{ (s, h) } S1$
and $\bigwedge m \ s' \ h'. m \leq n \wedge (s', h') \in S1 \implies \text{safe } m \text{ (Some } \Gamma) \ C2 \text{ (s', h') } S2$
shows $\text{safe } n \text{ (Some } \Gamma) \text{ (Cseq } C1 \ C2) \text{ (s, h) } S2$
 $\langle \text{proof} \rangle$

lemma *seq-safe*:

fixes $\Delta :: ('i, 'a, nat) \text{ cont}$
assumes $\text{safe } n \ \Delta \ C1 \text{ (s, h) } S1$
and $\bigwedge m \ s' \ h'. m \leq n \wedge (s', h') \in S1 \implies \text{safe } m \ \Delta \ C2 \text{ (s', h') } S2$
shows $\text{safe } n \ \Delta \text{ (Cseq } C1 \ C2) \text{ (s, h) } S2$
 $\langle \text{proof} \rangle$

theorem *seq-rule*:

fixes $\Delta :: ('i, 'a, nat) \text{ cont}$
assumes $\text{hoare-triple-valid } \Delta \ P \ C1 \ R$
and $\text{hoare-triple-valid } \Delta \ R \ C2 \ Q$
shows $\text{hoare-triple-valid } \Delta \ P \text{ (Cseq } C1 \ C2) \ Q$
 $\langle \text{proof} \rangle$

4.4.11 Frame rule

lemma *safe-frame-None*:

assumes $\text{safe } n \text{ (None :: ('i, 'a, nat) cont) } C \text{ (s, h) } S$
and $\text{Some } H = \text{Some } h \oplus \text{Some } hf0$
and $\text{bounded } H$
shows $\text{safe } n \text{ (None :: ('i, 'a, nat) cont) } C \text{ (s, H) (add-states } S \ \{(s'', hf0) \mid s''\}.$
 $\text{agrees } (- \text{ wrC } C) \ s \ s''\})$
 $\langle \text{proof} \rangle$

lemma *safe-frame-Some*:

assumes $\text{safe } n \text{ (Some } \Gamma) \ C \text{ (s, h) } S$
and $\text{Some } H = \text{Some } h \oplus \text{Some } hf0$
and $\text{bounded } H$
shows $\text{safe } n \text{ (Some } \Gamma) \ C \text{ (s, H) (add-states } S \ \{(s'', hf0) \mid s''\}.$
 $\text{agrees } (- \text{ wrC } C) \ s \ s''\})$
 $\langle \text{proof} \rangle$

lemma *safe-frame*:

fixes $\Delta :: ('i, 'a, nat) \text{ cont}$
assumes $\text{safe } n \ \Delta \ C \text{ (s, h) } S$
and $\text{Some } H = \text{Some } h \oplus \text{Some } hf0$
and $\text{bounded } H$

shows *safe* $n \Delta C (s, H) (add\text{-}states\ S \{(s'', hf0) \mid s''.\ agrees\ (-\ wrC\ C)\ s\ s''\})$
 $\langle proof \rangle$

theorem *frame-rule*:

fixes $\Delta :: ('i, 'a, nat)\ cont$

assumes *hoare-triple-valid* $\Delta P C Q$

and *disjoint* $(fvA\ R)\ (wrC\ C)$

and *precise* $P \vee precise\ R$

shows *hoare-triple-valid* $\Delta (Star\ P\ R)\ C (Star\ Q\ R)$

$\langle proof \rangle$

4.4.12 Consequence

theorem *consequence-rule*:

fixes $\Delta :: ('i, 'a, nat)\ cont$

assumes *hoare-triple-valid* $\Delta P' C Q'$

and *entails* $P P'$

and *entails* $Q' Q$

shows *hoare-triple-valid* $\Delta P C Q$

$\langle proof \rangle$

4.4.13 Existential

theorem *existential-rule*:

fixes $\Delta :: ('i, 'a, nat)\ cont$

assumes *hoare-triple-valid* $\Delta P C Q$

and $x \notin fvC\ C$

and $\bigwedge \Gamma. \Delta = Some\ \Gamma \implies x \notin fvA\ (invariant\ \Gamma)$

and *unambiguous* $P\ x$

shows *hoare-triple-valid* $\Delta (Exists\ x\ P)\ C (Exists\ x\ Q)$

$\langle proof \rangle$

4.4.14 While loops

inductive *leads-to-loop* **where**

leads-to-loop $b\ I\ \Sigma\ \sigma\ \sigma$

$| \llbracket leads\text{-}to\text{-}loop\ b\ I\ \Sigma\ \sigma\ \sigma' ; bdenot\ b\ (fst\ \sigma') ; \sigma'' \in \Sigma\ \sigma' \rrbracket \implies leads\text{-}to\text{-}loop\ b\ I\ \Sigma\ \sigma\ \sigma''$

definition *leads-to-loop-set* **where**

leads-to-loop-set $b\ I\ \Sigma\ \sigma = \{ \sigma' \mid \sigma'.\ leads\text{-}to\text{-}loop\ b\ I\ \Sigma\ \sigma\ \sigma' \}$

definition *trans- Σ* **where**

trans- Σ $b\ I\ \Sigma\ \sigma = Set.filter\ (\lambda\sigma. \neg bdenot\ b\ (fst\ \sigma))\ (leads\text{-}to\text{-}loop\text{-}set\ b\ I\ \Sigma\ \sigma)$

inductive-cases *red-while-cases*: *red* $(Cwhile\ b\ s)\ \sigma\ C'\ \sigma'$

inductive-cases *abort-while-cases*: *aborts* $(Cwhile\ b\ s)\ \sigma$

lemma *safe-while-None*:

assumes $\bigwedge \sigma m. \sigma, \sigma \models \text{And } I \text{ (Bool } b) \implies \text{bounded (snd } \sigma) \implies \text{safe } n \text{ (None$
 $:: ('i, 'a, \text{nat}) \text{ cont}) } C \sigma (\Sigma \sigma)$
and $\bigwedge \sigma \sigma'. \sigma, \sigma' \models \text{And } I \text{ (Bool } b) \implies \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') I$
and $(s, h), (s, h) \models I$
and $\text{leads-to-loop } b I \Sigma \sigma (s, h)$
and $\text{bounded } h$
shows $\text{safe } n \text{ (None } :: ('i, 'a, \text{nat}) \text{ cont}) (C\text{while } b C) (s, h) (\text{trans-}\Sigma b I \Sigma \sigma)$
 $\langle \text{proof} \rangle$

lemma *safe-while-Some*:

assumes $\bigwedge \sigma m. \sigma, \sigma \models \text{And } I \text{ (Bool } b) \implies \text{bounded (snd } \sigma) \implies \text{safe } n \text{ (Some$
 $\Gamma) } C \sigma (\Sigma \sigma)$
and $\bigwedge \sigma \sigma'. \sigma, \sigma' \models \text{And } I \text{ (Bool } b) \implies \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') I$
and $(s, h), (s, h) \models I$
and $\text{leads-to-loop } b I \Sigma \sigma (s, h)$
shows $\text{safe } n \text{ (Some } \Gamma) (C\text{while } b C) (s, h) (\text{trans-}\Sigma b I \Sigma \sigma)$
 $\langle \text{proof} \rangle$

lemma *safe-while*:

fixes $\Delta :: ('i, 'a, \text{nat}) \text{ cont}$
assumes $\bigwedge \sigma m. \sigma, \sigma \models \text{And } I \text{ (Bool } b) \implies \text{bounded (snd } \sigma) \implies \text{safe } n \Delta C \sigma$
 $(\Sigma \sigma)$
and $\bigwedge \sigma \sigma'. \sigma, \sigma' \models \text{And } I \text{ (Bool } b) \implies \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') I$
and $(s, h), (s, h) \models I$
and $\text{leads-to-loop } b I \Sigma \sigma (s, h)$
and $\text{bounded } h$
shows $\text{safe } n \Delta (C\text{while } b C) (s, h) (\text{trans-}\Sigma b I \Sigma \sigma)$
 $\langle \text{proof} \rangle$

lemma *leads-to-sat-inv-unary*:

assumes $\text{leads-to-loop } b I \Sigma \sigma \sigma'$
and $\bigwedge \sigma \sigma'. \sigma, \sigma' \models (\text{And } I \text{ (Bool } b)) \implies \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') I$
and $\sigma, \sigma \models I$
shows $\sigma', \sigma' \models I$
 $\langle \text{proof} \rangle$

theorem *while-rule2*:

fixes $\Delta :: ('i, 'a, \text{nat}) \text{ cont}$
assumes *unary* I
and *hoare-triple-valid* $\Delta (\text{And } I \text{ (Bool } b)) C I$
shows *hoare-triple-valid* $\Delta I (C\text{while } b C) (\text{And } I \text{ (Bool (Bnot } b)))$
 $\langle \text{proof} \rangle$

fun *iterate-sigma* $:: \text{nat} \implies \text{bexp} \implies ('i, 'a, \text{nat}) \text{ assertion} \implies ((\text{store} \times ('i, 'a) \text{ heap})$
 $\implies (\text{store} \times ('i, 'a) \text{ heap}) \text{ set}) \implies (\text{store} \times ('i, 'a) \text{ heap}) \implies (\text{store} \times ('i, 'a) \text{ heap})$
 set

where

iterate-sigma $0 b I \Sigma \sigma = \{\sigma\}$

| *iterate-sigma* (Suc n) b I Σ σ = ($\bigcup \sigma' \in \text{Set.filter } (\lambda\sigma. \text{bdenot } b \text{ (fst } \sigma)) \text{ (iterate-sigma } n \text{ b I } \Sigma \sigma). \Sigma \sigma'$)

lemma *union-of-iterate-sigma-is-leads-to-loop-set:*

assumes *leads-to-loop* b I Σ σ σ'
shows $\sigma' \in (\bigcup n. \text{iterate-sigma } n \text{ b I } \Sigma \sigma)$
<proof>

lemma *trans-included:*

trans- Σ b I Σ $\sigma \subseteq \text{Set.filter } (\lambda\sigma. \neg \text{bdenot } b \text{ (fst } \sigma)) (\bigcup n. \text{iterate-sigma } n \text{ b I } \Sigma \sigma)$
<proof>

lemma *iterate-sigma-low-all-sat-I-and-low:*

assumes $\bigwedge \sigma \sigma'. \sigma, \sigma' \models (\text{And } I \text{ (Bool } b)) \implies \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') (\text{And } I \text{ (Low } b))$
and $\sigma 1, \sigma 2 \models I$
and $\text{bdenot } b \text{ (fst } \sigma 1) = \text{bdenot } b \text{ (fst } \sigma 2)$
shows $\text{pair-sat } (\text{iterate-sigma } n \text{ b I } \Sigma \sigma 1) (\text{iterate-sigma } n \text{ b I } \Sigma \sigma 2) (\text{And } I \text{ (Low } b))$
<proof>

lemma *iterate-empty-later-empty:*

assumes $\text{iterate-sigma } n \text{ b I } \Sigma \sigma = \{\}$
and $m \geq n$
shows $\text{iterate-sigma } m \text{ b I } \Sigma \sigma = \{\}$
<proof>

lemma *all-same:*

assumes $\bigwedge \sigma \sigma'. \sigma, \sigma' \models (\text{And } I \text{ (Bool } b)) \implies \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') (\text{And } I \text{ (Low } b))$
and $\sigma 1, \sigma 2 \models I$
and $\text{bdenot } b \text{ (fst } \sigma 1) = \text{bdenot } b \text{ (fst } \sigma 2)$
and $x 1 \in \text{iterate-sigma } n \text{ b I } \Sigma \sigma 1$
and $x 2 \in \text{iterate-sigma } n \text{ b I } \Sigma \sigma 2$
shows $\text{bdenot } b \text{ (fst } x 1) = \text{bdenot } b \text{ (fst } x 2)$
<proof>

lemma *non-empty-at-most-once:*

assumes $\bigwedge \sigma \sigma'. \sigma, \sigma' \models (\text{And } I \text{ (Bool } b)) \implies \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') (\text{And } I \text{ (Low } b))$
and $\sigma, \sigma \models I$
and $\text{Set.filter } (\lambda\sigma. \neg \text{bdenot } b \text{ (fst } \sigma)) (\text{iterate-sigma } n 1 \text{ b I } \Sigma \sigma) \neq \{\}$
and $\text{Set.filter } (\lambda\sigma. \neg \text{bdenot } b \text{ (fst } \sigma)) (\text{iterate-sigma } n 2 \text{ b I } \Sigma \sigma) \neq \{\}$
shows $n 1 = n 2$
<proof>

lemma *one-non-empty-union*:

assumes $\bigwedge \sigma \sigma'. \sigma, \sigma' \models (\text{And } I (\text{Bool } b)) \implies \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') (\text{And } I (\text{Low } b))$
and $\sigma, \sigma \models I$
and $\text{Set.filter } (\lambda \sigma. \neg \text{bdenot } b (\text{fst } \sigma)) (\text{iterate-sigma } k \ b \ I \ \Sigma \ \sigma) \neq \{\}$
shows $\text{Set.filter } (\lambda \sigma. \neg \text{bdenot } b (\text{fst } \sigma)) (\bigcup n. \text{iterate-sigma } n \ b \ I \ \Sigma \ \sigma) = \text{Set.filter } (\lambda \sigma. \neg \text{bdenot } b (\text{fst } \sigma)) (\text{iterate-sigma } k \ b \ I \ \Sigma \ \sigma)$
 $\langle \text{proof} \rangle$

definition *not-set where*

$\text{not-set } b \ S = \text{Set.filter } (\lambda \sigma. \neg \text{bdenot } b (\text{fst } \sigma)) \ S$

lemma *union-exists-at-some-point-exactly*:

assumes $\bigwedge \sigma \sigma'. \sigma, \sigma' \models (\text{And } I (\text{Bool } b)) \implies \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') (\text{And } I (\text{Low } b))$
and $\sigma 1, \sigma 2 \models I$
and $\text{bdenot } b (\text{fst } \sigma 1) = \text{bdenot } b (\text{fst } \sigma 2)$
and $\text{Set.filter } (\lambda \sigma. \neg \text{bdenot } b (\text{fst } \sigma)) (\bigcup n. \text{iterate-sigma } n \ b \ I \ \Sigma \ \sigma 1) \neq \{\}$
and $\text{Set.filter } (\lambda \sigma. \neg \text{bdenot } b (\text{fst } \sigma)) (\bigcup n. \text{iterate-sigma } n \ b \ I \ \Sigma \ \sigma 2) \neq \{\}$
shows $\exists k. \text{not-set } b (\bigcup n. \text{iterate-sigma } n \ b \ I \ \Sigma \ \sigma 1) = \text{not-set } b (\text{iterate-sigma } k \ b \ I \ \Sigma \ \sigma 1) \wedge \text{not-set } b (\bigcup n. \text{iterate-sigma } n \ b \ I \ \Sigma \ \sigma 2) = \text{not-set } b (\text{iterate-sigma } k \ b \ I \ \Sigma \ \sigma 2)$
 $\langle \text{proof} \rangle$

theorem *while-rule1*:

fixes $\Delta :: ('i, 'a, \text{nat}) \text{ cont}$
assumes $\text{hoare-triple-valid } \Delta (\text{And } I (\text{Bool } b)) \ C (\text{And } I (\text{Low } b))$
shows $\text{hoare-triple-valid } \Delta (\text{And } I (\text{Low } b)) (\text{Cwhile } b \ C) (\text{And } I (\text{Bool } (\text{Bnot } b)))$
 $\langle \text{proof} \rangle$

lemma *entails-smallerI*:

assumes $\bigwedge s1 \ h1 \ s2 \ h2. (s1, h1), (s2, h2) \models A \implies (s1, h1), (s2, h2) \models B$
shows $\text{entails } A \ B$
 $\langle \text{proof} \rangle$

corollary *while-rule*:

fixes $\Delta :: ('i, 'a, \text{nat}) \text{ cont}$
assumes $\text{entails } P (\text{Star } P' \ R)$
and $\text{unary } P'$
and $\text{fv } A \ R \cap \text{wr } C \ C = \{\}$
and $\text{hoare-triple-valid } \Delta (\text{And } P' (\text{Bool } e)) \ C \ P'$
and $\text{hoare-triple-valid } \Delta (\text{And } P (\text{Bool } (\text{Band } e \ e'))) \ C (\text{And } P (\text{Low } (\text{Band } e \ e')))$
and $\text{precise } P' \vee \text{precise } R$
shows $\text{hoare-triple-valid } \Delta (\text{And } P (\text{Low } (\text{Band } e \ e'))) (\text{Cseq } (\text{Cwhile } (\text{Band } e \ e')))$

$e' \ C) \ (Cwhile \ e \ C) \ (And \ (Star \ P' \ R) \ (Bool \ (Bnot \ e)))$
 $\langle proof \rangle$

4.4.15 CommCSL is sound

theorem *soundness*:

assumes $\Delta \vdash \{P\} \ C \ \{Q\}$

shows $\Delta \models \{P\} \ C \ \{Q\}$

$\langle proof \rangle$

4.5 Corollaries

The two following corollaries express what proving a Hoare triple in CommCSL with no invariant (initially) guarantees, i.e., that if C is executed in two states that together satisfy the precondition P, then no execution will abort, and any pair of final states will satisfy together the postcondition Q.

This first corollary considers that the heap h1 is part of a larger execution with heap H1.

theorem *safety*:

assumes *hoare-triple-valid* $(None :: ('i, 'a, nat) \ cont) \ P \ C \ Q$

and $(s1, h1), (s2, h2) \models P$

and $Some \ H1 = Some \ h1 \oplus Some \ hf1 \wedge full\text{-}ownership \ (get\text{-}fh \ H1) \wedge no\text{-}guard \ H1$

— extend h1 to a normal state H1 without guards

and $Some \ H2 = Some \ h2 \oplus Some \ hf2 \wedge full\text{-}ownership \ (get\text{-}fh \ H2) \wedge no\text{-}guard \ H2$

— extend h2 to a normal state H2 without guards

shows $\bigwedge \sigma' \ C'. \ red\text{-}rtrans \ C \ (s1, \ normalize \ (get\text{-}fh \ H1)) \ C' \ \sigma' \implies \neg \ aborts \ C' \ \sigma'$

and $\bigwedge \sigma' \ C'. \ red\text{-}rtrans \ C \ (s2, \ normalize \ (get\text{-}fh \ H2)) \ C' \ \sigma' \implies \neg \ aborts \ C' \ \sigma'$

and $\bigwedge \sigma 1' \ \sigma 2'. \ red\text{-}rtrans \ C \ (s1, \ normalize \ (get\text{-}fh \ H1)) \ Cskip \ \sigma 1'$

$\implies \ red\text{-}rtrans \ C \ (s2, \ normalize \ (get\text{-}fh \ H2)) \ Cskip \ \sigma 2'$

$\implies (\exists \ h1' \ h2' \ H1' \ H2'. \ no\text{-}guard \ H1' \wedge full\text{-}ownership \ (get\text{-}fh \ H1') \wedge snd \ \sigma 1' = \ normalize \ (get\text{-}fh \ H1') \wedge Some \ H1' = Some \ h1' \oplus Some \ hf1$

$\wedge no\text{-}guard \ H2' \wedge full\text{-}ownership \ (get\text{-}fh \ H2') \wedge snd \ \sigma 2' = \ normalize \ (get\text{-}fh \ H2') \wedge Some \ H2' = Some \ h2' \oplus Some \ hf2$

$\wedge (fst \ \sigma 1', \ h1'), (fst \ \sigma 2', \ h2') \models Q$

$\langle proof \rangle$

lemma *neutral-add*:

$Some \ h = Some \ h \oplus Some \ (Map.empty, \ None, \ (\lambda_. \ None))$

$\langle proof \rangle$

This second corollary considers that the heap h1 is the only execution that

matters, and thus it ignores any frame. It corresponds to Corollary 4.5 in the paper.

corollary *safety-no-frame*:

assumes *hoare-triple-valid* (*None* :: (*'i*, *'a*, *nat*) *cont*) *P C Q*
and (*s1*, *H1*), (*s2*, *H2*) \models *P*

and *full-ownership* (*get-fh H1*) \wedge *no-guard H1*
and *full-ownership* (*get-fh H2*) \wedge *no-guard H2*

shows $\bigwedge \sigma' C'. \text{red-rtrans } C (s1, \text{normalize } (\text{get-fh } H1)) C' \sigma' \implies \neg \text{aborts } C'$
 σ'

and $\bigwedge \sigma' C'. \text{red-rtrans } C (s2, \text{normalize } (\text{get-fh } H2)) C' \sigma' \implies \neg \text{aborts } C'$
 σ'

and $\bigwedge \sigma 1' \sigma 2'. \text{red-rtrans } C (s1, \text{normalize } (\text{get-fh } H1)) Cskip \sigma 1'$
 $\implies \text{red-rtrans } C (s2, \text{normalize } (\text{get-fh } H2)) Cskip \sigma 2'$

$\implies (\exists H1' H2'. \text{no-guard } H1' \wedge \text{full-ownership } (\text{get-fh } H1') \wedge \text{snd } \sigma 1' = \text{normalize } (\text{get-fh } H1')$

$\wedge \text{no-guard } H2' \wedge \text{full-ownership } (\text{get-fh } H2') \wedge \text{snd } \sigma 2' = \text{normalize } (\text{get-fh } H2')$

$\wedge (\text{fst } \sigma 1', H1'), (\text{fst } \sigma 2', H2') \models Q$

$\langle \text{proof} \rangle$

end

theory *NonInterference*

imports *Soundness*

begin

In this file, we prove two non-interference theorems, based on the soundness of CommCSL.

fun *low-list* **where**

low-list [] = *Bool Btrue*

| *low-list* (*v* # *q*) = *And* (*LowExp* (*Evar v*)) (*low-list q*)

lemma *low-listE*:

assumes (*s1*, *h1*), (*s2*, *h2*) \models *low-list l*

and *x* \in *set l*

shows *s1 x* = *s2 x*

$\langle \text{proof} \rangle$

lemma *low-listI*:

assumes $\bigwedge x. x \in \text{set } l \implies s1\ x = s2\ x$

shows (*s1*, *h1*), (*s2*, *h2*) \models *low-list l*

$\langle \text{proof} \rangle$

corollary *non-interference*:

assumes (*None* :: (*'i*, *'a*, *nat*) *cont*) \vdash {*And P* (*low-list In*)} *C* {*low-list Out*}

and *red-rtrans* *C* (*s1*, *normalize* (*get-fh H1*)) *Cskip* (*s1'*, *h1'*)

and *red-rtrans* *C* (*s2*, *normalize* (*get-fh H2*)) *Cskip* (*s2'*, *h2'*)

and $\bigwedge x. x \in \text{set } In \implies s1\ x = s2\ x$

and $x \in \text{set } Out$
and $(s1, H1), (s2, H2) \models P$
and $\text{full-ownership } (\text{get-fh } H1) \wedge \text{no-guard } H1$
and $\text{full-ownership } (\text{get-fh } H2) \wedge \text{no-guard } H2$
shows $s1' x = s2' x$
 $\langle \text{proof} \rangle$

definition *heapify* **where**

$\text{heapify } h = (\lambda l. \text{apply-opt } (\lambda v. (\text{pwrite}, v)) (h \ l), \text{None}, \lambda-. \text{None})$

lemma *heapify-properties*:

$\text{full-ownership } (\text{get-fh } (\text{heapify } h))$
 $\text{no-guard } (\text{heapify } h)$
 $\text{normalize } (\text{get-fh } (\text{heapify } h)) = h$
 $\langle \text{proof} \rangle$

corollary *non-interference-no-precondition*:

assumes $(\text{None} :: ('i, 'a, \text{nat}) \text{cont}) \vdash \{\text{low-list } In\} C \{\text{low-list } Out\}$
and $\text{red-rtrans } C (s1, h1) C\text{skip } (s1', h1')$
and $\text{red-rtrans } C (s2, h2) C\text{skip } (s2', h2')$
and $\bigwedge x. x \in \text{set } In \implies s1 x = s2 x$
and $x \in \text{set } Out$
shows $s1' x = s2' x$
 $\langle \text{proof} \rangle$

end

References

- [1] M. Eilers, T. Dardinier, and P. Müller. CommCSL: Proving information flow security for concurrent programs using abstract commutativity, 2022.
- [2] V. Vafeiadis. Concurrent separation logic and operational semantics. In M. W. Mislove and J. Ouaknine, editors, *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS 2011, Pittsburgh, PA, USA, May 25-28, 2011*, volume 276 of *Electronic Notes in Theoretical Computer Science*, pages 335–351. Elsevier, 2011.