

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

September 13, 2023

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	15
2.1	Language Syntax and Semantics	15
2.1.1	Semantics of expressions	16
2.1.2	Semantics of commands	17
2.1.3	Abort semantics	17
2.2	Useful Definitions and Results	18
3	CommCSL	21
3.1	Assertion Language	22
3.2	Rules of the Logic	30
4	Soundness of CommCSL	32
4.1	Abstract Commutativity	32
4.2	Consistency	39
4.3	Safety and Hoare Triples	46
4.3.1	Preliminaries	46
4.3.2	Safety	48
4.3.3	Useful results about safety	51
4.3.4	Hoare triples	53
4.4	Soundness of the Rules	53
4.4.1	Skip	53
4.4.2	Assign	54
4.4.3	Alloc	54
4.4.4	Write	55
4.4.5	Read	55
4.4.6	Share	56
4.4.7	Atomic	57
4.4.8	Parallel	59
4.4.9	If	61
4.4.10	Sequential composition	62
4.4.11	Frame rule	62
4.4.12	Consequence	63
4.4.13	Existential	63
4.4.14	While loops	63
4.4.15	CommCSL is sound	67
4.5	Corollaries	67

1 State Model

1.1 Partial Heaps

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)*

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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

lemma *compatible-maps-empty*:
 compatible-maps f h (Map.empty)
 \langle *proof* \rangle

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

lemma *add-heaps-asso*:

$(h1 ++ h2) ++ h3 = h1 ++ (h2 ++ h3)$
<proof>

lemma *compatible-maps-same*:
 assumes *compatible-maps* (=) *ha hb*
 and *ha x = Some y*
 shows $(ha ++ hb) x = Some y$
<proof>

lemma *compatible-maps-refl*:
 compatible-maps (=) *h h*
<proof>

lemma *map-invo*:
 $h ++ h = h$
<proof>

lemma *included-then-compatible-maps*:
 assumes *map-included h1 h*
 and *map-included h2 h*
 shows *compatible-maps* (=) *h1 h2*
<proof>

lemma *commut-charact*:
 assumes *compatible-maps* (=) *h1 h2*
 shows $h1 ++ h2 = h2 ++ h1$
<proof>

end

1.2 Fractional Permissions

theory *PosRat*
 imports *Main HOL.Rat*
begin

typedef *prat* = { *r :: rat* | *r. r > 0* } *<proof>*

setup-lifting *type-definition-prat*

lift-definition *prwrite :: prat is 1* *<proof>*

lift-definition *half :: prat is 1 / 2* *<proof>*

lift-definition *pgte :: prat ⇒ prat ⇒ bool is (≥)* *<proof>*

lift-definition *pgt :: prat ⇒ prat ⇒ bool is (>)* *<proof>*

lift-definition *lt :: prat ⇒ prat ⇒ bool is (<)* *<proof>*

lift-definition *pmult :: prat ⇒ prat ⇒ prat is (*)* *<proof>*

lift-definition *padd :: prat ⇒ prat ⇒ prat is (+)* *<proof>*

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 $half-between-0-1$:

$pgt\ pwrite\ half$
 $\langle proof \rangle$

lemma $pgt-implies-pgte$:

assumes $pgt\ a\ b$
shows $pgte\ a\ b$
 $\langle proof \rangle$

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 $pgt\ (padd\ a\ c)\ (padd\ b\ d)$
<proof>

lemma *pmult-distr*:
 $pmult\ a\ (padd\ b\ c) = padd\ (pmult\ a\ b)\ (pmult\ a\ c)$
<proof>

lemma *pmult-comm*:
 $pmult\ a\ b = pmult\ b\ a$
{proof}

lemma *pmult-special*:
 $pmult\ pwrite\ x = x$
{proof}

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

lemma *pinv-double-half*:
 $pmult\ half\ (pinv\ p) = pinv\ (padd\ p\ p)$
{proof}

lemma *pinv-inverts*:
assumes $pgte\ a\ b$
shows $pgte\ (pinv\ b)\ (pinv\ a)$
{proof}

lemma *pinv-pmult-ok*:
 $pmult\ p\ (pinv\ p) = pwrite$
{proof}

lemma *pinv-pwrite*:
 $pinv\ pwrite = pwrite$
{proof}

lemma *pmin-pmax*:
assumes $pgte\ x\ (pmin\ a\ b)$
shows $x = pmin\ (pmax\ x\ a)\ (pmax\ x\ b)$
{proof}

lemma *pmin-sum*:
 $padd\ (pmin\ a\ b)\ c = pmin\ (padd\ a\ c)\ (padd\ b\ c)$
{proof}

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

theory *FractionalHeap*

imports *Main PosRat PartialMap*

begin

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

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 $\bigwedge l p p'. h l = \text{Some } p \wedge h' l = \text{Some } p' \Longrightarrow \text{pgte } p \text{ write } (\text{padd } (\text{fst } p) (\text{fst } p'))$

and $\bigwedge l p p'. h l = \text{Some } p \wedge h' l = \text{Some } p' \Longrightarrow \text{snd } p = \text{snd } p'$

shows *compatible-fract-heaps* h h'

<proof>

lemma *compatible-fract-heapsE*:

assumes *compatible-fract-heaps* h h'

and h l = Some p \wedge h' l = Some p'

shows pgte p write (padd (fst p) (fst p'))

and snd p = snd p'

<proof>

lemma *compatible-fract-heaps-comm*:

assumes *compatible-fract-heaps* h h'

shows *compatible-fract-heaps* $h' h$
<proof>

The following definition 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

add-fh $h h' l = fadd-options (h l) (h' l)$

definition *full-ownership* :: $(l, 'v)$ *fract-heap* $\Rightarrow bool$ **where**

full-ownership $h \longleftrightarrow (\forall l p. h l = Some p \longrightarrow fst p = pwrite)$

lemma *full-ownershipI*:

assumes $\bigwedge l p. h l = Some p \implies fst p = pwrite$

shows *full-ownership* h

<proof>

fun *apply-opt* **where**

apply-opt $f None = None$

| *apply-opt* $f (Some x) = Some (f x)$

definition *normalize* :: $(l, 'v)$ *fract-heap* $\Rightarrow (l \rightarrow 'v)$ **where**

normalize $h l = apply-opt snd (h l)$

lemma *normalize-eq*:

normalize $h l = None \longleftrightarrow h l = None$

normalize $h l = Some v \longleftrightarrow (\exists p. h l = Some (p, v))$ (**is** $?A \longleftrightarrow ?B$)

<proof>

definition *fpdom* **where**

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

lemma *compatible-then-dom-disjoint*:

assumes *compatible-fract-heaps* $h1 h2$

shows $dom h1 \cap fpdom h2 = \{\}$

and $dom h2 \cap fpdom h1 = \{\}$

<proof>

lemma *compatible-dom-sum*:

assumes *compatible-fract-heaps* $h1 h2$

shows $dom (add-fh h1 h2) = dom h1 \cup dom h2$ (**is** $?A = ?B$)

<proof>

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 $add\text{-}fh\ (a(x \mapsto p))\ b = (add\text{-}fh\ a\ b)(x \mapsto p)$
 ⟨*proof*⟩

end

1.4 Extended Heaps

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 $f\text{-}heap = (loc, val)\ fract\text{-}heap$

type-synonym $'a\ gs\text{-}heap = (prat \times 'a\ multiset)\ option$

type-synonym $('i, 'a)\ gu\text{-}heap = 'i \Rightarrow 'a\ list\ option$

type-synonym $('i, 'a)\ heap = f\text{-}heap \times 'a\ gs\text{-}heap \times ('i, 'a)\ gu\text{-}heap$

type-synonym $var = string$

type-synonym $normal\text{-}heap = (nat \rightarrow nat)$

type-synonym $store = (var \Rightarrow nat)$

fun $get\text{-}fh$ **where** $get\text{-}fh\ x = fst\ x$

fun $get\text{-}gs$ **where** $get\text{-}gs\ x = fst\ (snd\ x)$

fun $get\text{-}gu$ **where** $get\text{-}gu\ x = snd\ (snd\ x)$

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 $compatible :: ('i, 'a)\ heap \Rightarrow ('i, 'a)\ heap \Rightarrow bool$ (**infixl** $##$ 60) **where**
 $h ## h' \iff compatible\text{-}fract\text{-}heaps\ (get\text{-}fh\ h)\ (get\text{-}fh\ h') \wedge (\forall k. get\text{-}gu\ h\ k = None \vee get\text{-}gu\ h'\ k = None)$
 $\wedge (\forall p\ p'. get\text{-}gs\ h = Some\ p \wedge get\text{-}gs\ h' = Some\ p' \implies pgte\ pwrite\ (padd\ (fst\ p)\ (fst\ p')))$

lemma $compatibleI$:

assumes $compatible\text{-}fract\text{-}heaps\ (get\text{-}fh\ h)\ (get\text{-}fh\ h')$

and $\bigwedge k. get\text{-}gu\ h\ k = None \vee get\text{-}gu\ h'\ k = None$

and $\bigwedge p\ p'. get\text{-}gs\ h = Some\ p \wedge get\text{-}gs\ h' = Some\ p' \implies pgte\ pwrite\ (padd\ (fst\ p)\ (fst\ p'))$

shows $h ## h'$

⟨*proof*⟩

fun $add\text{-}gu\text{-}single$ **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 $\bigwedge k. h\ k = \text{None} \vee h'\ k = \text{None}$

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'*)

lemma *add-gs-cancellative*:

assumes *add-gs* *a* *x* = *add-gs* *b* *x*

shows *a* = *b*

<proof>

lemma *add-gs-comm*:

add-gs *a* *b* = *add-gs* *b* *a*

<proof>

lemma *compatible-fheaps-comm*:

assumes *compatible-fract-heaps* *a* *b*

shows *add-fh* *a* *b* = *add-fh* *b* *a*

<proof>

fun *plus* :: ('*i*, '*a*) *heap option* ⇒ ('*i*, '*a*) *heap option* ⇒ ('*i*, '*a*) *heap option* (**infixl** ⊕ 63) **where**

None ⊕ - = *None*

| - ⊕ *None* = *None*

| *Some* *h1* ⊕ *Some* *h2* = (if *h1* ## *h2* then *Some* (*add-fh* (*get-fh* *h1*) (*get-fh* *h2*), *add-gs* (*get-gs* *h1*) (*get-gs* *h2*), *add-gu* (*get-gu* *h1*) (*get-gu* *h2*)) else *None*)

lemma *plus-extract*:

assumes *Some* *x* = *Some* *a* ⊕ *Some* *b*

shows *get-fh* *x* = *add-fh* (*get-fh* *a*) (*get-fh* *b*)

and *get-gs* *x* = *add-gs* (*get-gs* *a*) (*get-gs* *b*)

and *get-gu* *x* = *add-gu* (*get-gu* *a*) (*get-gu* *b*)

<proof>

lemma *compatible-eq*:

Some *a* ⊕ *Some* *b* = *None* ⇔ ¬ *a* ## *b*

<proof>

lemma *compatible-comm*:

$a \#\# b \longleftrightarrow b \#\# a$
<proof>

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 $a = b$
<proof>

lemma *plus-comm*:

$a \oplus b = b \oplus a$
<proof>

lemma *asso2*:

assumes $Some\ a \oplus Some\ b = Some\ ab$
and $\neg b \#\# c$
shows $\neg ab \#\# c$
<proof>

lemma *plus-extract-point-fh*:

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

lemma *asso1*:

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

lemma *simpler-asso*:

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

lemma *plus-asso*:

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

definition *larger* :: $('i, 'a)\ heap \Rightarrow ('i, 'a)\ heap \Rightarrow bool$ (**infixl** \succeq 55) **where**

$a \succeq b \longleftrightarrow (\exists c. Some\ a = Some\ b \oplus Some\ c)$

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 $\text{get-gs } a = \text{Some } (pwrite, sargs)$
and $\text{Some } h = \text{Some } a \oplus \text{Some } b$
shows $\text{get-gs } h = \text{Some } (pwrite, sargs)$
 ⟨*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 $\text{get-gu } h \ k = \text{Some } uargs$
 ⟨*proof*⟩

lemma *smaller-more-compatible*:
assumes $a \#\# b$
and $a \succeq c$
shows $c \#\# b$
 ⟨*proof*⟩

lemma *equiv-sum-get-fh*:
assumes $\text{get-fh } a = \text{get-fh } a'$
and $\text{get-fh } b = \text{get-fh } b'$
and $\text{Some } x = \text{Some } a \oplus \text{Some } b$
and $\text{Some } x' = \text{Some } a' \oplus \text{Some } b'$
shows $\text{get-fh } x = \text{get-fh } x'$
 ⟨*proof*⟩

lemma *addition-cancellative*:
assumes $\text{Some } a = \text{Some } b \oplus \text{Some } c$
and $\text{Some } a = \text{Some } b' \oplus \text{Some } c$
shows $b = b'$
 ⟨*proof*⟩

lemma *addition-cancellative3*:
assumes $\text{Some } x = \text{Some } a \oplus \text{Some } b \oplus \text{Some } c$
and $\text{Some } x = \text{Some } a' \oplus \text{Some } b \oplus \text{Some } c$
shows $a = a'$
 ⟨*proof*⟩

lemma *larger3*:
assumes $\text{Some } x = \text{Some } a \oplus \text{Some } b \oplus \text{Some } c$
shows $x \succeq b$
 $\langle \text{proof} \rangle$

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

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

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 Some x = Some a ⊕ Some b
  shows dom (get-fh x) = dom (get-fh a) ∪ dom (get-fh b)
  ⟨proof⟩

lemma dom-three-sum:
  assumes Some x = Some a ⊕ Some b ⊕ Some c
  shows dom (get-fh x) = dom (get-fh a) ∪ dom (get-fh b) ∪ dom (get-fh c)
  ⟨proof⟩

lemma addition-smaller-domain:
  assumes Some a = Some b ⊕ Some c
  shows dom (get-fh b) ⊆ dom (get-fh a)
  ⟨proof⟩

lemma one-value-sum-same:
  assumes Some x = Some a ⊕ Some b
    and get-fh a l = Some (π, v)
  shows ∃ π'. get-fh x l = Some (π', v)
  ⟨proof⟩

lemma compatible-last-two:
  assumes Some x = Some a ⊕ Some b ⊕ Some c
  shows b ## c
  ⟨proof⟩

lemma add-fh-map-empty:
  add-fh h Map.empty = h
  ⟨proof⟩

end

```

2 Imperative Concurrent Language

This file defines the syntax and semantics of a concurrent programming language, 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/cslsound/>).

```

theory Lang
imports Main StateModel
begin

```

2.1 Language Syntax and Semantics

```

type-synonym state = store × 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

inductive

$aborts :: cmd \Rightarrow state \Rightarrow bool$

where

$aborts\text{-}Seq[intro]: aborts C1 \sigma \Longrightarrow aborts (Cseq C1 C2) \sigma$
 $| aborts\text{-}Atomic[intro]: \llbracket red\text{-}rtrans C \sigma C' \sigma'; aborts C' \sigma' \rrbracket \Longrightarrow aborts (Catomic C) \sigma$
 $| aborts\text{-}Par1[intro]: aborts C1 \sigma \Longrightarrow aborts (Cpar C1 C2) \sigma$
 $| aborts\text{-}Par2[intro]: aborts C2 \sigma \Longrightarrow aborts (Cpar C1 C2) \sigma$
 $| aborts\text{-}Read[intro]: edenot E (fst \sigma) \notin dom (snd \sigma) \Longrightarrow aborts (Cread x E) \sigma$
 $| aborts\text{-}Write[intro]: edenot E (fst \sigma) \notin dom (snd \sigma) \Longrightarrow aborts (Cwrite E E') \sigma$
 $| aborts\text{-}Free[intro]: edenot E (fst \sigma) \notin dom (snd \sigma) \Longrightarrow aborts (Cdispose E) \sigma$

inductive-cases *abort-atomic-cases: aborts* (*Catomic C*) σ

2.2 Useful Definitions and Results

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

primrec

$fvE :: exp \Rightarrow var\ set$

where

$fvE (Evar\ v) = \{v\}$
 $| fvE (Enum\ n) = \{\}$
 $| fvE (Eplus\ e1\ e2) = (fvE\ e1 \cup fvE\ e2)$

primrec

$fvB :: bexp \Rightarrow var\ set$

where

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

primrec

$fvC :: cmd \Rightarrow var\ set$

where

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

primrec

$wrC :: cmd \Rightarrow var\ set$

where

$wrC (Cskip) = \{\}$
 $| wrC (Cassign\ v\ E) = \{v\}$
 $| wrC (Cread\ v\ E) = \{v\}$
 $| wrC (Cwrite\ E1\ E2) = \{\}$
 $| wrC (Calloc\ v\ E) = \{v\}$
 $| wrC (Cdispose\ E) = \{\}$
 $| wrC (Cseq\ C1\ C2) = (wrC\ C1 \cup wrC\ C2)$
 $| wrC (Cpar\ C1\ C2) = (wrC\ C1 \cup wrC\ C2)$
 $| wrC (Cif\ B\ C1\ C2) = (wrC\ C1 \cup wrC\ C2)$

| $wrC (Cwhile\ B\ C) = (wrC\ C)$
| $wrC (Catomic\ C) = (wrC\ C)$

primrec

$subE :: var \Rightarrow exp \Rightarrow exp \Rightarrow exp$

where

$subE\ x\ E\ (Evar\ y) = (if\ x = y\ then\ E\ else\ Evar\ y)$
| $subE\ x\ E\ (Enum\ n) = Enum\ n$
| $subE\ x\ E\ (Eplus\ e1\ e2) = Eplus\ (subE\ x\ E\ e1)\ (subE\ x\ E\ e2)$

primrec

$subB :: var \Rightarrow exp \Rightarrow bexp \Rightarrow bexp$

where

$subB\ x\ E\ (Beq\ e1\ e2) = Beq\ (subE\ x\ E\ e1)\ (subE\ x\ E\ e2)$
| $subB\ x\ E\ (Band\ b1\ b2) = Band\ (subB\ x\ E\ b1)\ (subB\ x\ E\ b2)$
| $subB\ x\ E\ (Bnot\ b) = Bnot\ (subB\ x\ E\ b)$
| $subB\ x\ E\ Btrue = Btrue$

Basic properties of substitutions:

lemma *subE-assign*:

$edenot\ (subE\ x\ E\ e)\ s = edenot\ e\ (s(x := edenot\ E\ s))$
⟨proof⟩

lemma *subB-assign*:

$bdenot\ (subB\ x\ E\ b)\ s = bdenot\ b\ (s(x := edenot\ E\ s))$
⟨proof⟩

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

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

lemma *skip-simps[simp]*:

$\neg\ red\ Cskip\ \sigma\ C'\ \sigma'$
 $\neg\ aborts\ Cskip\ \sigma$
⟨proof⟩

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*:

$agrees\ (A \cup B)\ s\ s' \longleftrightarrow agrees\ A\ s\ s' \wedge agrees\ B\ s\ s'$
⟨proof⟩

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

lemma *agreesI*:

assumes $\bigwedge x. x \in X \implies s x = s' x$
shows *agrees* $X s s'$
 ⟨*proof*⟩

lemma *red-properties:*

red $C \sigma C' \sigma' \implies \text{fv}C C' \subseteq \text{fv}C C \wedge \text{wr}C C' \subseteq \text{wr}C C \wedge \text{agrees} (- \text{wr}C C) (\text{fst} \sigma')$ (*fst* σ)

red-rtrans $C \sigma C' \sigma' \implies \text{fv}C C' \subseteq \text{fv}C C \wedge \text{wr}C C' \subseteq \text{wr}C C \wedge \text{agrees} (- \text{wr}C C) (\text{fst} \sigma') (\text{fst} \sigma)$

⟨*proof*⟩

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

lemma *exp-agrees:* *agrees* (*fvE* E) $s s' \implies \text{edenot } E s = \text{edenot } E s'$
 ⟨*proof*⟩

lemma *bexp-agrees:*

agrees (*fvB* B) $s s' \implies \text{bdenot } B s = \text{bdenot } B s'$
 ⟨*proof*⟩

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

red $C \sigma C' \sigma' \implies x \notin \text{fv}C C \implies \text{fst } \sigma x = \text{fst } \sigma' x$

red-rtrans $C \sigma C' \sigma' \implies x \notin \text{fv}C C \implies \text{fst } \sigma x = \text{fst } \sigma' x$

⟨*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 $C \sigma C' \sigma' \implies (\forall s h. \text{agrees } X (\text{fst } \sigma) s \wedge \text{snd } \sigma = h \wedge \text{fv}C C \subseteq X \longrightarrow$

$(\exists s' h'. \text{red } C (s, h) C' (s', h') \wedge \text{agrees } X (\text{fst } \sigma') s' \wedge \text{snd } \sigma' = h')$

red-rtrans $C \sigma C' \sigma' \implies (\forall s h. \text{agrees } X (\text{fst } \sigma) s \wedge \text{snd } \sigma = h \wedge \text{fv}C C \subseteq X$

\longrightarrow

$(\exists s' h'. \text{red-rtrans } C (s, h) C' (s', h') \wedge \text{agrees } X (\text{fst } \sigma') s' \wedge \text{snd } \sigma' = h')$

⟨*proof*⟩

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

red $C \sigma C' \sigma' \implies \forall X s. \text{agrees } X (\text{fst } \sigma) s \longrightarrow \text{snd } \sigma = h \longrightarrow \text{fv}C C \subseteq X \longrightarrow$

$(\exists s' h'. \text{red } C (s, h) C' (s', h') \wedge \text{agrees } X (\text{fst } \sigma') s' \wedge \text{snd } \sigma' = h')$

⟨*proof*⟩

lemma *aborts-agrees:*

assumes *aborts C* σ
and *agrees (fvC C) (fst σ) s*
and *snd $\sigma = h$*
shows *aborts C (s, h)*
 \langle *proof* \rangle

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

lemma *agrees-update*:
assumes $a \notin S$
shows *agrees S s (s(a := v))*
 \langle *proof* \rangle

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

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

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

end

3 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 $'a \text{ precondition} = \{ \text{pre} :: ('a \Rightarrow 'a \Rightarrow \text{bool}) \mid \text{pre}. \forall a b. \text{pre } a b \longrightarrow (\text{pre } b a \wedge \text{pre } a a) \}$
 \langle *proof* \rangle

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

typedef (*i*, *a*) *indexed-precondition* = { *pre* :: (*i* ⇒ *a* ⇒ *a* ⇒ *bool*) | *pre*. ∀ *a* *b*
k. *pre* *k* *a* *b* ⇒ (*pre* *k* *b* *a* ∧ *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* ∧ *Rep-indexed-precondition* *pre* *k* *a* *a*

⟨*proof*⟩

type-synonym *'a list-exp* = *store* ⇒ *'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* (- * - 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* ⇒ *'v* (*i*, *a*, *v*) *assertion* *store* ⇒ *'v*

| *SharedGuard* *prat* *store* ⇒ *'a multiset*

| *UniqueGuard* *'i 'a list-exp*

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

| *NoGuard*

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

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

| *Step*: [*PRE-shared-simpler* *spre* *a* *b* ; *spre* *xa* *xb*] ⇒ *PRE-shared-simpler* *spre*
 (*a* + {# *xa* #}) (*b* + {# *xb* #})

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

PRE-unique *upre* *uargs* *uargs'* ⇔ *length* *uargs* = *length* *uargs'* ∧ (∀ *i*. *i* ≥ 0 ∧
i < *length* *uargs'* ⇒ *upre* (*uargs* ! *i*) (*uargs'* ! *i*))

fun *hyper-sat* :: (*store* × (*i*, *a*) *heap*) ⇒ (*store* × (*i*, *a*) *heap*) ⇒ (*'i*, *'a*, *nat*)

assertion \Rightarrow *bool* $(-, - \models - [51, 65, 66] 50)$ **where**

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

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

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

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

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

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

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

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

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

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

lemma *sat-PreUniqueE*:

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

shows $\exists \text{uargs } \text{uargs}' . (\forall k. \text{get-gu } h \ k = \text{Some } (\text{uargs } k)) \wedge (\forall k. \text{get-gu } h' \ k = \text{Some } (\text{uargs}' k)) \wedge (\forall k. \text{PRE-unique } (\text{Rep-indexed-precondition } \text{upre } k) (\text{uargs } k) (\text{uargs}' k))$
 ⟨proof⟩

lemma *decompose-heap-triple*:

$h = (\text{get-fh } h, \text{get-gs } h, \text{get-gu } h)$
 ⟨proof⟩

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

lemma *depends-only-onI*:

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

definition *fvS* :: $(\text{store} \Rightarrow 'v) \Rightarrow \text{var set}$ **where**

$\text{fvS } e = (\text{SOME } S. \text{depends-only-on } e \ S)$

lemma *fvSE*:

assumes $\text{agrees } (\text{fvS } e) \ s \ s'$
shows $e \ s = e \ s'$
 ⟨proof⟩

fun *fvA* :: $('i, 'a, 'v)$ *assertion* $\Rightarrow \text{var set}$ **where**

$\text{fvA } (\text{Bool } b) = \text{fvB } b$
 | $\text{fvA } (\text{And } A \ B) = \text{fvA } A \cup \text{fvA } B$
 | $\text{fvA } (\text{Star } A \ B) = \text{fvA } A \cup \text{fvA } B$
 | $\text{fvA } (\text{Low } e) = \text{fvB } e$
 | $\text{fvA } \text{Emp} = \{\}$
 | $\text{fvA } (\text{PointsTo } v \ va \ vb) = \text{fvE } v \cup \text{fvE } vb$
 | $\text{fvA } (\text{Exists } x \ A) = \text{fvA } A - \{x\}$
 | $\text{fvA } (\text{SharedGuard } - \ e) = \text{fvS } e$
 | $\text{fvA } (\text{UniqueGuard } - \ e) = \text{fvS } e$
 | $\text{fvA } (\text{View view } A \ e) = \text{fvA } A \cup \text{fvS } e$
 | $\text{fvA } (\text{PreSharedGuards } -) = \{\}$
 | $\text{fvA } (\text{PreUniqueGuards } -) = \{\}$
 | $\text{fvA } \text{EmptyFullGuards} = \{\}$
 | $\text{fvA } (\text{LowExp } x) = \text{fvE } x$
 | $\text{fvA } (\text{Imp } b \ A) = \text{fvB } b \cup \text{fvA } A$

definition *subS* :: $\text{var} \Rightarrow \text{exp} \Rightarrow (\text{store} \Rightarrow 'v) \Rightarrow (\text{store} \Rightarrow 'v)$ **where**

$\text{subS } x \ E \ e = (\lambda s. e \ (s(x := \text{edenot } E \ s)))$

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-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 (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⟩

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 inv

definition view-function-of-inv :: ('i, 'a, nat) single-context \Rightarrow bool **where**
 view-function-of-inv Γ \longleftrightarrow (\forall (h :: ('i, 'a) heap) (h' :: ('i, 'a) heap) s. (s, h),
 h) \models invariant Γ \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

(- \vdash {-} - {-} [51,0,0] 81) **where**

RuleSkip: $\Delta \vdash \{P\}$ *Cskip* $\{P\}$

| *RuleAssign*: $\llbracket \bigwedge \Gamma. \Delta = \text{Some } \Gamma \Longrightarrow x \notin \text{fvA } (\text{invariant } \Gamma) ; \text{collect-existentials } P \cap \text{fvE } E = \{\} \rrbracket \Longrightarrow \Delta \vdash \{\text{subA } x E P\}$ *Cassign* $x E \{P\}$

| *RuleNew*: $\llbracket x \notin \text{fvE } E ; \bigwedge \Gamma. \Delta = \text{Some } \Gamma \Longrightarrow x \notin \text{fvA } (\text{invariant } \Gamma) \wedge \text{view-function-of-inv } \Gamma \rrbracket \Longrightarrow \Delta \vdash \{\text{Emp}\}$ *Calloc* $x E \{\text{PointsTo } (\text{Evar } x) \text{pwrite } E\}$

| *RuleWrite*: $\llbracket \bigwedge \Gamma. \Delta = \text{Some } \Gamma \Longrightarrow \text{view-function-of-inv } \Gamma ; v \notin \text{fvE } \text{loc} \rrbracket \Longrightarrow \Delta \vdash \{\text{Exists } v (\text{PointsTo } \text{loc } \text{pwrite } (\text{Evar } v))\}$ *Cwrite* $\text{loc } E \{\text{PointsTo } \text{loc } \text{pwrite } E\}$

| $\llbracket \bigwedge \Gamma. \Delta = \text{Some } \Gamma \Longrightarrow x \notin \text{fvA } (\text{invariant } \Gamma) \wedge \text{view-function-of-inv } \Gamma ; x \notin \text{fvE } E \cup \text{fvE } e \rrbracket \Longrightarrow$

$\Delta \vdash \{\text{PointsTo } E \pi e\}$ *Cread* $x E \{\text{And } (\text{PointsTo } E \pi e) (\text{Bool } (\text{Beq } (\text{Evar } x) e))\}$

| *RuleShare*: $\llbracket \Gamma = () \text{view} = f, \text{abstract-view} = \alpha, \text{saction} = \text{sact}, \text{uaction} = \text{uact}, \text{invariant} = J \rrbracket ; \text{all-axioms } \alpha \text{ sact spre uact upre} ;$

$\text{Some } \Gamma \vdash \{\text{Star } P \text{EmptyFullGuards}\} C \{\text{Star } Q (\text{And } (\text{PreSharedGuards } (\text{Abs-precondition spre})) (\text{PreUniqueGuards } (\text{Abs-indexed-precondition upre})))\} ;$

$\text{view-function-of-inv } \Gamma ; \text{unary } J ; \text{precise } J ; \text{wf-indexed-precondition upre} ; \text{wf-precondition spre} ; x \notin \text{fvA } J ;$

$\text{no-guard-assertion } (\text{Star } P (\text{LowView } (\alpha \circ f) J x)) \rrbracket \Longrightarrow \text{None } \vdash \{\text{Star } P (\text{LowView } (\alpha \circ f) J x)\} C \{\text{Star } Q (\text{LowView } (\alpha \circ f) J x)\}$

| *RuleAtomicUnique*: $\llbracket \Gamma = () \text{view} = f, \text{abstract-view} = \alpha, \text{saction} = \text{sact}, \text{uaction} = \text{uact}, \text{invariant} = J \rrbracket ;$

$\text{no-guard-assertion } P ; \text{no-guard-assertion } Q ;$

$\text{None } \vdash \{\text{Star } P (\text{View } f J (\lambda s. s x))\} C \{\text{Star } Q (\text{View } f J (\lambda s. \text{uact } \text{index } (s x) (\text{map-to-arg } (s \text{uarg}))))\} ;$

*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

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 *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 *upre i' uarg' uarg'*

and *upre i uarg uarg*

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

<proof>

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

PRE-shared spre sargs sargs' ⟷ (∃ ms. image-mset fst ms = sargs ∧ image-mset
snd ms = sargs' ∧ (∀ x ∈# ms. spre (fst x) (snd x)))

lemma *PRE-shared-same-size*:

assumes *PRE-shared spre sargs sargs'*

shows *size sargs = size sargs'*

<proof>

definition *is-Unique* :: ('i, 'a, 'b) action ⇒ bool **where**

is-Unique a ⟷ ¬ *is-Shared a*

definition *is-Unique-i* :: 'i ⇒ ('i, 'a, 'b) action ⇒ bool **where**

is-Unique-i i a ⟷ *is-Unique a* ∧ *get-i a = i*

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

possible-sequence sargs uargs s ⟷ ((∀ i. uargs i = map get-u (filter (*is-Unique-i*
i) s)) ∧ sargs = image-mset get-s (filter-mset *is-Shared* (mset s)))

lemma *possible-sequenceI*:

assumes $\bigwedge i. uargs i = map\ get-u\ (filter\ (is-Unique-i\ i)\ s)$

and *sargs = image-mset get-s (filter-mset is-Shared (mset s))*

shows *possible-sequence sargs uargs s*

<proof>

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

remove-at-index - [] = []

| *remove-at-index* 0 (x # xs) = xs

| *remove-at-index* (Suc n) (x # xs) = x # (*remove-at-index* n xs)

lemma *remove-at-index*:

assumes $n < length\ l$

shows $length\ (remove-at-index\ n\ l) = length\ l - 1$

and $i \geq 0 \wedge i < n \implies remove-at-index\ n\ l ! i = l ! i$

and $i \geq n \wedge i < length\ l - 1 \implies remove-at-index\ n\ l ! i = l ! (i + 1)$

<proof>

fun *insert-at* :: nat ⇒ 'd ⇒ 'd list ⇒ '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 $n \leq length\ l$

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 $i > n \wedge i < \text{length } l + 1 \implies \text{insert-at } n \ x \ l ! i = l ! (i - 1)$
 $\langle \text{proof} \rangle$

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 $a = b$
 $\langle \text{proof} \rangle$

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

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

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

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

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 $\text{tl } (\text{map } \text{get-u } (\text{filter } P \ s)) = \text{map } \text{get-u } (\text{filter } P \ (\text{remove-at-index } i \ s))$
 $\langle \text{proof} \rangle$

definition *tail-kth* **where**
 $\text{tail-kth } \text{uargs } k = \text{uargs}(k := \text{tl } (\text{uargs } k))$

lemma *exists-index-in-sequence-unique*:
assumes $a = \text{hd } (\text{uargs } k)$

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 $length\ s = length\ s'$
and $\bigwedge i. i \geq 0 \wedge i < length\ s \implies abstract-pre\ spre\ upre\ (s\ !\ i)\ (s'\ !\ i)$
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 ⇒ 'a ⇒ 'v) ⇒ ('i ⇒ 'v ⇒ 'b ⇒ 'v) ⇒ 'v ⇒ ('i, 'a, 'b) *action*
list ⇒ '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 ∈# sargs*
shows $\exists \text{sarg}'. \text{sarg}' \in \# \text{sargs}' \wedge \text{spre sarg sarg}' \wedge \text{PRE-shared spre (sargs - \{\# sarg \#\}) (sargs}' - \{\# \text{sarg}' \#\})$
<proof>

lemma *exists-aligned-sequence:*
assumes *possible-sequence sargs uargs s*
and *possible-sequence sargs' uargs' s'*

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

shows $\exists s''. \text{possible-sequence sargs}' \text{uargs}' s'' \wedge \text{PRE-sequence spre upre s s''}$
<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* α *sact spre uact upre*
shows $\alpha (\text{compute sact uact } v0\ s) = \alpha (\text{compute sact uact } v0'\ s')$
 <proof>

lemma *PRE-sequence-same-abstract:*

assumes *PRE-sequence spre upre s s'*
and $\alpha v0 = \alpha v0'$
and *all-axioms α sact spre uact upre*
shows $\alpha (\text{compute sact uact } v0 s) = \alpha (\text{compute sact uact } v0' s')$
<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 $\bigwedge k. \text{PRE-unique } (upre k) (uargs k) (uargs' k)$
and *all-axioms α 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 $\bigwedge k. \text{PRE-unique } (upre k) (uargs k) (uargs' k)$

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

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

inductive *reachable-value* :: $('v \Rightarrow 'a \Rightarrow 'v) \Rightarrow ('i \Rightarrow 'v \Rightarrow 'b \Rightarrow 'v) \Rightarrow 'v \Rightarrow 'a$
multiset $\Rightarrow ('i \Rightarrow 'b \text{ list}) \Rightarrow 'v \Rightarrow \text{bool}$ **where**

Self: reachable-value sact uact v0 {#} ($\lambda k. []$) v0
| *SharedStep: reachable-value sact uact v0 sargs uargs v1 \implies reachable-value sact uact v0 (sargs + {# sarg #}) uargs (sact v1 sarg)*
| *UniqueStep: reachable-value sact uact v0 sargs uargs v1 \implies reachable-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>

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  $\bigwedge k. \text{PRE-unique } (\text{upre } k) (\text{uargs } k) (\text{uargs}' k)$ 
    and  $\alpha v0 = \alpha v0'$ 
    and all-axioms  $\alpha$  sact spre uact upre
  shows  $\alpha v = \alpha v'$ 
<proof>

end

```

4.2 Consistency

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

lemma *reachableI*:

```

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

```

lemma *reachableE*:

```

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

```

definition *all-guards* :: $(i, 'a)$ *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  $\text{get-gs } h = \text{None}$ 
  and  $\bigwedge k. \text{get-gu } h k = \text{None}$ 
  shows no-guard h

```

<proof>

definition *semi-consistent* :: ('i, 'a, 'v) *single-context* \Rightarrow 'v \Rightarrow ('i, 'a) *heap* \Rightarrow *bool*

where

semi-consistent Γ v0 h \longleftrightarrow *all-guards* h \wedge *reachable* Γ v0 h

lemma *semi-consistentE*:

assumes *semi-consistent* Γ v0 h

shows \exists sargs uargs. *get-gs* h = *Some* (pwrite, sargs) \wedge (\forall k. *get-gu* h k = *Some* (uargs k))

\wedge *reachable-value* (saction Γ) (uaction Γ) v0 sargs uargs (*view* Γ (*normalize* (*get-fh* h)))

<proof>

lemma *semi-consistentI*:

assumes *all-guards* h

and *reachable* Γ v0 h

shows *semi-consistent* Γ 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. *get-gu* h k \neq *None*

and \exists v. *get-gs* h = *Some* (pwrite, v)

shows *all-guards* h

<proof>

lemma *all-guards-same*:

assumes *all-guards* a

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

shows *all-guards* h

<proof>

definition *empty-unique where*

empty-unique - = *None*

definition *remove-guards* :: ('i, 'a) *heap* \Rightarrow ('i, 'a) *heap* **where**

remove-guards h = (*get-fh* h, *None*, *empty-unique*)

lemma *remove-guards-smaller*:

h \succeq *remove-guards* h

<proof>

lemma *no-guard-remove*:

assumes $\text{Some } a = \text{Some } b \oplus \text{Some } c$
and $\text{no-guard } c$
shows $\text{get-gs } a = \text{get-gs } b$
and $\text{get-gu } a = \text{get-gu } b$
 $\langle \text{proof} \rangle$

lemma *full-guard-comp-then-no*:
assumes $a \#\# b$
and $\text{all-guards } a$
shows $\text{no-guard } b$
 $\langle \text{proof} \rangle$

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

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

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

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**
 $\text{pair-sat } S S' Q \iff (\forall \sigma \sigma'. \sigma \in S \wedge \sigma' \in S' \longrightarrow \sigma, \sigma' \models Q)$

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 $\text{pair-sat } S S' Q$
 $\langle \text{proof} \rangle$

lemma *pair-sat-smallerI*:
assumes $\bigwedge \sigma \sigma'. \sigma \in S \wedge \sigma' \in S' \implies \sigma, \sigma' \models Q$
shows $\text{pair-sat } S S' Q$
 $\langle \text{proof} \rangle$

lemma *pair-satE*:
assumes $\text{pair-sat } S S' Q$
and $(s, h) \in S \wedge (s', h') \in S'$
shows $(s, h), (s', h') \models Q$
 $\langle \text{proof} \rangle$

definition *add-states* :: $(\text{store} \times ('i, 'a) \text{ heap}) \text{ set} \Rightarrow (\text{store} \times ('i, 'a) \text{ heap}) \text{ set} \Rightarrow$
 $(\text{store} \times ('i, 'a) \text{ heap}) \text{ set}$ **where**

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

lemma *add-states-sat-star*:
assumes *pair-sat SA SA' A*
and *pair-sat SB SB' B*
shows *pair-sat (add-states SA SB) (add-states SA' SB') (Star A B)*
<proof>

lemma *add-states-subset*:
assumes $S1 \subseteq S1'$
shows $add\text{-}states\ S1\ S2 \subseteq add\text{-}states\ S1'\ S2$
<proof>

lemma *add-states-comm*:
 $add\text{-}states\ S1\ S2 = add\text{-}states\ S2\ S1$
<proof>

lemma *magic-lemma*:
assumes $Some\ x1 = Some\ a1 \oplus Some\ j1$
and $Some\ x2 = Some\ a2 \oplus Some\ j2$
and $(s1, x1), (s2, x2) \models Star\ A\ J$
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 $full\text{-}ownership\ (get\text{-}fh\ h) \wedge no\text{-}guard\ h$
and $full\text{-}ownership\ (get\text{-}fh\ h') \wedge no\text{-}guard\ h'$
and $normalize\ (get\text{-}fh\ h) = normalize\ (get\text{-}fh\ h')$
shows $h = h'$
<proof>

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

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

lemma *no-guard-smaller*:
assumes $a \succeq b$

shows *remove-guards* $a \succeq$ *remove-guards* b
 ⟨*proof*⟩

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

lemma *no-guard-map-empty-compatible*:

assumes *no-guard* a
and *get-fh* $b = \text{Map.empty}$
shows $a \#\# b$
 ⟨*proof*⟩

lemma *no-guard-add-empty-is-add*:

assumes *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 *no-guard* $h \wedge$ *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 *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* $\Gamma (\text{view } \Gamma (\text{FractionalHeap.normalize } (\text{get-fh } h))) (\text{add-empty-guards } h)$
 ⟨*proof*⟩

lemma *no-guards-remove-same*:

assumes *no-guard* h
shows $h = \text{remove-guards } (\text{add-empty-guards } h)$
 ⟨*proof*⟩

lemma *no-guards-remove*:

no-guard $h \longleftrightarrow h = \text{remove-guards } h$
 ⟨*proof*⟩

definition *add-sguard-to-no-guard* :: ('i, 'a) heap \Rightarrow prat \Rightarrow 'a multiset \Rightarrow ('i, 'a) heap **where**

add-sguard-to-no-guard $h \pi ms = (\text{get-fh } h, \text{Some } (\pi, ms), (\lambda_. \text{None}))$

lemma *get-fh-add-sguard*:

get-fh (add-sguard-to-no-guard h π ms) = *get-fh* h
{proof}

lemma *add-sguard-as-sum*:

assumes *no-guard* h
shows *Some* (add-sguard-to-no-guard h π ms) = *Some* h \oplus *Some* (*Map.empty*,
Some (π , ms), (λ -. *None*))
{proof}

definition *add-uguard-to-no-guard* :: 'i \Rightarrow ('i, 'a) heap \Rightarrow 'a list \Rightarrow ('i, 'a) heap
where

add-uguard-to-no-guard k h l = (*get-fh* h, *None*, (λ -. *None*))(k := *Some* l)

lemma *get-fh-add-uguard*:

get-fh (add-uguard-to-no-guard k h l) = *get-fh* h
{proof}

lemma *prove-sum*:

assumes a ## b
and $\bigwedge x$. *Some* x = *Some* a \oplus *Some* b \implies x = y
shows *Some* y = *Some* a \oplus *Some* b
{proof}

lemma *add-uguard-as-sum*:

assumes *no-guard* h
shows *Some* (add-uguard-to-no-guard k h l) = *Some* h \oplus *Some* (*Map.empty*,
None, (λ -. *None*))(k := *Some* l)
{proof}

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
{proof}

lemma *decompose-guard-remove-easy*:

Some h = *Some* (*remove-guards* h) \oplus *Some* (*Map.empty*, *get-gs* h, *get-gu* h)
{proof}

lemma *all-guards-no-guard-propagates*:

assumes *all-guards* x

and $\text{Some } x = \text{Some } a \oplus \text{Some } b$
and $\text{no-guard } a$
shows $\text{all-guards } b$
 $\langle \text{proof} \rangle$

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

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

fun $\text{add-pwrite-option where}$
 $\text{add-pwrite-option } \text{None} = \text{None}$
 $|\ \text{add-pwrite-option } (\text{Some } x) = \text{Some } (\text{pwrite}, x)$

definition $\text{denormalize} :: \text{normal-heap} \Rightarrow ('i, 'a) \text{ heap where}$
 $\text{denormalize } H = ((\lambda l. \text{add-pwrite-option } (H \ l)), \text{None}, (\lambda -. \text{None}))$

lemma $\text{denormalize-properties}$:
shows $\text{no-guard } (\text{denormalize } H)$
and $\text{full-ownership } (\text{get-fh } (\text{denormalize } H))$
and $\text{normalize } (\text{get-fh } (\text{denormalize } H)) = H$
and $\text{full-ownership } (\text{get-fh } h) \wedge \text{no-guard } h \implies \text{denormalize } (\text{normalize } (\text{get-fh } h)) = h$
and $\text{full-ownership } (\text{get-fh } h) \implies \text{denormalize } (\text{normalize } (\text{get-fh } h)) = \text{remove-guards } h$
 $\langle \text{proof} \rangle$

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

lemma $\text{no-guard-then-sat-star}$:
assumes $\text{no-guard } h \wedge \text{no-guard } h'$
and $(s, h), (s', h') \models Q$
shows $(s, \text{add-sguard-to-no-guard } h \ \pi \ (\text{ms } s)), (s', \text{add-sguard-to-no-guard } h' \ \pi$

$(ms\ s') \models \text{Star } Q\ (\text{SharedGuard } \pi\ ms)$
 $\langle \text{proof} \rangle$

end

4.3 Safety and Hoare Triples

theory *Safety*
imports *Guards*
begin

4.3.1 Preliminaries

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

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

lemma *sat-invI*:

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

and $\text{no-guard } hj$

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

$\langle \text{proof} \rangle$

s and s' can differ on variables outside of vars, does not change anything.
upper-fvs S 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**
 $\text{upper-fvs } S\ \text{vars} \iff (\forall s\ s'\ h. (s, h) \in S \wedge \text{agrees vars } s\ s' \implies (s', h) \in S)$

Only need to agree on vars

definition *upperize* **where**

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

definition *close-var* **where**

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

lemma *upper-fvsI*:

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

shows $\text{upper-fvs } S\ \text{vars}$

$\langle \text{proof} \rangle$

lemma *pair-sat-comm*:

assumes $\text{pair-sat } S\ S'\ A$

shows $\text{pair-sat } S'\ S\ A$

$\langle \text{proof} \rangle$

lemma *in-upperize*:

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

<proof>

lemma *upper-fvs-upperize*:
 upper-fvs (upperize S vars) vars
<proof>

lemma *upperize-larger*:
 $S \subseteq \text{upperize } S \text{ vars}$
<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*:
 $(s', h) \in \text{close-var } S \ x \longleftrightarrow (\exists s \ v. (s, h) \in S \wedge s' = s(x := v))$ (**is** $?A \longleftrightarrow ?B$)
<proof>

lemma *pair-sat-close-var*:
 assumes $x \notin \text{fvA } A$
 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 $x \notin \text{fvA } A$
 shows *pair-sat (close-var S x) (close-var S' x) A*
<proof>

lemma *close-var-subset*:
 $S \subseteq \text{close-var } S \ x$
<proof>

lemma *upper-fvs-close-vars*:
 upper-fvs (close-var S x) (- {x})
<proof>

lemma *sat-inv-agrees*:
 assumes *sat-inv s hj Γ*
 and *agrees (fvA (invariant Γ)) s s'*
 shows *sat-inv s' hj Γ*
<proof>

lemma *abort-iff-fvC*:
 assumes *agrees (fvC C) s s'*
 shows *aborts C (s, h) \longleftrightarrow aborts C (s', h)*
<proof>

lemma *view-function-of-invE*:

assumes *view-function-of-inv* Γ
and *sat-inv* $s\ h\ \Gamma$
and $(h' :: ('i, 'a)\ \text{heap}) \succeq h$
shows *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**
no-abort $\text{None}\ C\ s\ h \longleftrightarrow (\forall\ hf\ H.\ \text{Some}\ H = \text{Some}\ h \oplus \text{Some}\ hf \wedge \text{full-ownership}\ (\text{get-fh}\ H) \wedge \text{no-guard}\ H$
 $\longrightarrow \neg\ \text{aborts}\ C\ (s, \text{normalize}\ (\text{get-fh}\ H)))$
 $|$ *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 $\bigwedge (hf :: ('i, 'a)\ \text{heap})\ (H :: ('i, 'a)\ \text{heap}).\ \text{Some}\ H = \text{Some}\ h \oplus \text{Some}\ hf \wedge \Delta = \text{None} \wedge \text{full-ownership}\ (\text{get-fh}\ H) \wedge \text{no-guard}\ H \Longrightarrow \neg\ \text{aborts}\ C\ (s, \text{normalize}\ (\text{get-fh}\ H))$
and $\bigwedge H\ hf\ hj\ v0\ \Gamma.\ \Delta = \text{Some}\ \Gamma \wedge \text{Some}\ H = \text{Some}\ h \oplus \text{Some}\ hj \oplus \text{Some}\ hf \wedge \text{full-ownership}\ (\text{get-fh}\ H) \wedge \text{semi-consistent}\ \Gamma\ v0\ H \wedge \text{sat-inv}\ s\ hj\ \Gamma$
 $\Longrightarrow \neg\ \text{aborts}\ C\ (s, \text{normalize}\ (\text{get-fh}\ H))$
shows *no-abort* $\Delta\ C\ s\ (h :: ('i, 'a)\ \text{heap})$
 $\langle \text{proof} \rangle$

lemma *no-abortSomeI*:

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

lemma *no-abortNoneI*:

assumes $\bigwedge (hf :: ('i, 'a)\ \text{heap})\ (H :: ('i, 'a)\ \text{heap}).\ \text{Some}\ H = \text{Some}\ h \oplus \text{Some}\ hf \wedge \text{full-ownership}\ (\text{get-fh}\ H) \wedge \text{no-guard}\ H \Longrightarrow \neg\ \text{aborts}\ C\ (s, \text{normalize}\ (\text{get-fh}\ H))$
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 $\text{Some}\ H = \text{Some}\ h \oplus \text{Some}\ hf \Longrightarrow \Delta = \text{None} \Longrightarrow \text{full-ownership}\ (\text{get-fh}\ H) \Longrightarrow \text{no-guard}\ H \Longrightarrow \neg\ \text{aborts}\ C\ (s, \text{normalize}\ (\text{get-fh}\ H))$
and $\Delta = \text{Some}\ \Gamma \Longrightarrow \text{Some}\ H = \text{Some}\ h \oplus \text{Some}\ hj \oplus \text{Some}\ hf \Longrightarrow \text{sat-inv}\ s\ hj\ \Gamma \Longrightarrow \text{full-ownership}\ (\text{get-fh}\ H) \Longrightarrow \text{semi-consistent}\ \Gamma\ v0\ H$

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

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

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

| *safe* (*Suc* *n*) (*Some* Γ) *C* (*s*, *h*) *S* \longleftrightarrow (*C* = *Cskip* \longrightarrow (*s*, *h*) \in *S*) \wedge *no-abort* (*Some* Γ) *C* *s* *h* \wedge
 ($\forall H$ *hf* *C'* *s'* *h'* *hj* *v0*. *Some* *H* = *Some* *h* \oplus *Some* *hj* \oplus *Some* *hf* \wedge *full-ownership* (*get-fh* *H*) \wedge *semi-consistent* Γ *v0* *H* \wedge *sat-inv* *s* *hj* Γ
 \wedge *red* *C* (*s*, *normalize* (*get-fh* *H*)) *C'* (*s'*, *h'*)
 \longrightarrow ($\exists h''$ *H'* *hj'*. *full-ownership* (*get-fh* *H'*) \wedge *semi-consistent* Γ *v0* *H'* \wedge *sat-inv* *s'* *hj'* Γ
 \wedge *h'* = *normalize* (*get-fh* *H'*) \wedge *Some* *H'* = *Some* *h''* \oplus *Some* *hj'* \oplus *Some* *hf* \wedge *safe* *n* (*Some* Γ) *C'* (*s'*, *h''*) *S*)

lemma *safeNoneI*:

assumes *C* = *Cskip* \implies (*s*, *h*) \in *S*
and *no-abort* *None* *C* *s* *h*
and $\bigwedge H$ *hf* *C'* *s'* *h'*. *Some* *H* = *Some* *h* \oplus *Some* *hf* \wedge *full-ownership* (*get-fh* *H*) \wedge *no-guard* *H* \wedge *red* *C* (*s*, *normalize* (*get-fh* *H*)) *C'* (*s'*, *h'*)
 \implies ($\exists h''$ *H'*. *full-ownership* (*get-fh* *H'*) \wedge *no-guard* *H'* \wedge *h'* = *normalize* (*get-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* Γ) *C* *s* *h*
and $\bigwedge H$ *hf* *C'* *s'* *h'* *hj* *v0*. *Some* *H* = *Some* *h* \oplus *Some* *hj* \oplus *Some* *hf* \wedge *full-ownership* (*get-fh* *H*)
 \wedge *semi-consistent* Γ *v0* *H* \wedge *sat-inv* *s* *hj* Γ \wedge *red* *C* (*s*, *normalize* (*get-fh* *H*))
C' (*s'*, *h'*)
 \implies ($\exists h''$ *H'* *hj'*. *full-ownership* (*get-fh* *H'*) \wedge *semi-consistent* Γ *v0* *H'* \wedge *sat-inv* *s'* *hj'* Γ
 \wedge *h'* = *normalize* (*get-fh* *H'*) \wedge *Some* *H'* = *Some* *h''* \oplus *Some* *hj'* \oplus *Some* *hf* \wedge *safe* *n* (*Some* Γ) *C'* (*s'*, *h''*) *S*)
shows *safe* (*Suc* *n*) (*Some* Γ) *C* (*s*, *h* :: (*'i*, *'a*) *heap*) *S*

$\langle \text{proof} \rangle$

lemma *safeI*:

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

assumes $C = \text{Cskip} \implies (s, h) \in S$

and $\text{no-abort } \Delta \ C \ s \ h$

and $\bigwedge H \ hf \ C' \ s' \ h'. \ \Delta = \text{None} \implies \text{Some } H = \text{Some } h \oplus \text{Some } hf \wedge$
full-ownership $(\text{get-fh } H) \wedge \text{no-guard } H \wedge \text{red } C \ (s, \text{normalize } (\text{get-fh } H)) \ C' \ (s',$
 $h')$

$\implies (\exists h'' \ H'. \ \text{full-ownership } (\text{get-fh } H') \wedge \text{no-guard } H' \wedge h' = \text{normalize } (\text{get-fh}$
 $H') \wedge \text{Some } H' = \text{Some } h'' \oplus \text{Some } hf \wedge \text{safe } n \ (\text{None} :: ('i, 'a, \text{nat}) \text{ cont}) \ C' \ (s',$
 $h'') \ S)$

and $\bigwedge H \ hf \ C' \ s' \ h' \ hj \ v0 \ \Gamma. \ \Delta = \text{Some } \Gamma \implies \text{Some } H = \text{Some } h \oplus \text{Some}$
 $hj \oplus \text{Some } hf \wedge \text{full-ownership } (\text{get-fh } H)$

$\wedge \text{semi-consistent } \Gamma \ v0 \ H \wedge \text{sat-inv } s \ hj \ \Gamma \wedge \text{red } C \ (s, \text{normalize } (\text{get-fh } H))$
 $C' \ (s', h')$

$\implies (\exists h'' \ H' \ hj'. \ \text{full-ownership } (\text{get-fh } H') \wedge \text{semi-consistent } \Gamma \ v0 \ H' \wedge \text{sat-inv}$
 $s' \ hj' \ \Gamma$

$\wedge h' = \text{normalize } (\text{get-fh } H') \wedge \text{Some } H' = \text{Some } h'' \oplus \text{Some } hj' \oplus \text{Some } hf \wedge$
 $\text{safe } n \ (\text{Some } \Gamma) \ C' \ (s', h'') \ S)$

shows $\text{safe } (\text{Suc } n) \ \Delta \ C \ (s, h :: ('i, 'a) \text{ heap}) \ S$

$\langle \text{proof} \rangle$

lemma *safeSomeAltI*:

assumes $C = \text{Cskip} \implies (s, h) \in S$

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

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

and $\bigwedge H \ hf \ C' \ s' \ h' \ hj \ v0. \ \text{Some } H = \text{Some } h \oplus \text{Some } hj \oplus \text{Some } hf \wedge$
full-ownership $(\text{get-fh } H)$

$\wedge \text{semi-consistent } \Gamma \ v0 \ H \wedge \text{sat-inv } s \ hj \ \Gamma \implies \text{red } C \ (s, \text{normalize } (\text{get-fh}$
 $H)) \ C' \ (s', h')$

$\implies (\exists h'' \ H' \ hj'. \ \text{full-ownership } (\text{get-fh } H') \wedge \text{semi-consistent } \Gamma \ v0 \ H' \wedge \text{sat-inv}$
 $s' \ hj' \ \Gamma$

$\wedge h' = \text{normalize } (\text{get-fh } H') \wedge \text{Some } H' = \text{Some } h'' \oplus \text{Some } hj' \oplus \text{Some } hf \wedge$
 $\text{safe } n \ (\text{Some } \Gamma) \ C' \ (s', h'') \ S)$

shows $\text{safe } (\text{Suc } n) \ (\text{Some } \Gamma) \ C \ (s, h :: ('i, 'a) \text{ heap}) \ S$

$\langle \text{proof} \rangle$

lemma *safeSomeE*:

assumes $\text{safe } (\text{Suc } n) \ (\text{Some } \Gamma) \ C \ (s, h :: ('i, 'a) \text{ heap}) \ S$

shows $C = \text{Cskip} \implies (s, h) \in S$

and $\text{no-abort } (\text{Some } \Gamma) \ C \ s \ h$

and $\text{Some } H = \text{Some } h \oplus \text{Some } hj \oplus \text{Some } hf \implies \text{full-ownership } (\text{get-fh } H)$

$\implies \text{semi-consistent } \Gamma \ v0 \ H \implies \text{sat-inv } s \ hj \ \Gamma \implies \text{red } C \ (s, \text{normalize}$
 $(\text{get-fh } H)) \ C' \ (s', h')$

$\implies (\exists h'' \ H' \ hj'. \ \text{full-ownership } (\text{get-fh } H') \wedge \text{semi-consistent } \Gamma \ v0 \ H' \wedge \text{sat-inv}$

$s' \text{ hj}' \Gamma$
 $\wedge h' = \text{normalize}(\text{get-fh } H') \wedge \text{Some } H' = \text{Some } h'' \oplus \text{Some } \text{hj}' \oplus \text{Some } \text{hf} \wedge$
 $\text{safe } n (\text{Some } \Gamma) C' (s', h'') S$
 $\langle \text{proof} \rangle$

lemma *safeNoneE*:

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

lemma *safeNoneE-bis*:

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

4.3.3 Useful results about safety

lemma *no-abort-larger*:

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

lemma *safe-larger-set-aux*:

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

lemma *safe-larger-set*:

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

lemma *safe-smaller-aux*:

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

lemma *safe-smaller*:

assumes $m \leq n$
 and *safe* $n \Delta C \sigma S$
shows *safe* $m \Delta C \sigma S$
<proof>

lemma *safe-free-vars-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-fvs* $S vars$
 and $\bigwedge \Gamma. \Delta = Some \Gamma \implies \text{agrees } (fvA \text{ (invariant } \Gamma)) s0 s1$
shows *safe* $n \Delta C (s1, h) S$
<proof>

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-fvs* $S vars$
shows *safe* $n (None :: ('i, 'a, nat) cont) C (s', h) S$
<proof>

lemma *safe-free-vars-Some*:

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

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-fvs* $S vars$
 and $\bigwedge \Gamma. \Delta = Some \Gamma \implies \text{agrees } (fvA \text{ (invariant } \Gamma)) s s'$
shows *safe* $n \Delta C (s', h) S$
<proof>

4.3.4 Hoare triples

definition *hoare-triple-valid* :: ('i, 'a, nat) cont ⇒ ('i, 'a, nat) assertion ⇒ cmd
 ⇒ ('i, 'a, nat) assertion ⇒ bool
 (- ⊨ {-} - {-} [51,0,0] 81) **where**
hoare-triple-valid Γ P C Q ⇔ (∃Σ. (∀σ n. σ, σ ⊨ P → safe n Γ C σ (Σ σ))
 ∧
 (∀σ σ'. σ, σ' ⊨ P → pair-sat (Σ σ) (Σ σ') 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* Γ 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* Γ P C Q
 ⟨proof⟩

lemma *hoare-triple-validE*:

assumes *hoare-triple-valid* Γ P C Q
shows $\exists \Sigma. (\forall \sigma n. \sigma, \sigma \models P \implies \text{safe } n \Gamma C \sigma (\Sigma \sigma)) \wedge$
 $(\forall \sigma \sigma'. \sigma, \sigma' \models P \implies \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') Q)$
 ⟨proof⟩

lemma *hoare-triple-valid-simplerE*:

assumes *hoare-triple-valid* Γ P C Q
and $\sigma, \sigma' \models P$
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

theory *Soundness*

imports *Safety AbstractCommutativity*

begin

4.4.1 Skip

lemma *safe-skip*:

fixes Δ :: ('i, 'a, nat) cont
assumes $(s, h) \in S$
shows *safe* n Δ Cskip (s, h) S
 ⟨proof⟩

theorem *rule-skip*:
hoare-triple-valid Γ P C *skip* P
 ⟨*proof*⟩

4.4.2 Assign

inductive-cases *red-assign-cases*: *red* (*Cassign* x E) σ C' σ'
inductive-cases *aborts-assign-cases*: *aborts* (*Cassign* x E) σ

lemma *safe-assign*:
fixes $\Delta :: ('i, 'a, nat)$ *cont*
assumes $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies x \notin \text{fv}A$ (*invariant* Γ)
shows *safe* m Δ (*Cassign* x E) (s, h) $\{ (s(x := \text{edenot } E s), h) \}$
 ⟨*proof*⟩

theorem *assign-rule*:
fixes $\Delta :: ('i, 'a, nat)$ *cont*
assumes $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies x \notin \text{fv}A$ (*invariant* Γ)
and *collect-existentials* $P \cap \text{fv}E = \{ \}$
shows *hoare-triple-valid* Δ (*subA* x E P) (*Cassign* x E) P
 ⟨*proof*⟩

4.4.3 Alloc

inductive-cases *red-alloc-cases*: *red* (*Calloc* x E) σ C' σ'
inductive-cases *aborts-alloc-cases*: *aborts* (*Calloc* x E) σ

lemma *safe-new-None*:
safe n (*None* $:: ('i, 'a, nat)$ *cont*) (*Calloc* x E) $(s, (\text{Map.empty}, gs, gu))$ $\{ (s(x := a), (\text{Map.empty}(a \mapsto (\text{pwrite}, \text{edenot } E s)), gs, gu)) \mid a. \text{True} \}$
 ⟨*proof*⟩

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

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

theorem *new-rule*:

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

assumes $x \notin fvE E$

and $\bigwedge \Gamma. \Delta = Some \Gamma \implies x \notin fvA (invariant \Gamma) \wedge view-function-of-inv \Gamma$

shows *hoare-triple-valid* $\Delta Emp (Calloc x E) (PointsTo (Evar x) pwrite E)$

<proof>

4.4.4 Write

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

inductive-cases *aborts-write-cases*: *aborts* $(Cwrite x E) \sigma$

lemma *safe-write-None*:

assumes $fh (edenot loc s) = Some (pwrite, v)$

shows *safe n* $(None :: ('i, 'a, nat) cont) (Cwrite loc E) (s, (fh, gs, gu)) \{ (s, (fh(edenot loc s \mapsto (pwrite, edenot E s)), gs, gu)) \}$

<proof>

lemma *safe-write-Some*:

assumes $fh (edenot loc s) = Some (pwrite, v)$

and *view-function-of-inv* Γ

shows *safe n* $(Some \Gamma) (Cwrite loc E) (s, (fh, gs, gu)) \{ (s, (fh(edenot loc s \mapsto (pwrite, edenot E s)), gs, gu)) \}$

<proof>

lemma *safe-write*:

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

assumes $fh (edenot loc s) = Some (pwrite, v)$

and $\bigwedge \Gamma. \Delta = Some \Gamma \implies view-function-of-inv \Gamma$

shows *safe n* $\Delta (Cwrite loc E) (s, (fh, gs, gu)) \{ (s, (fh(edenot loc s \mapsto (pwrite, edenot E s)), gs, gu)) \}$

<proof>

theorem *write-rule*:

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

assumes $\bigwedge \Gamma. \Delta = Some \Gamma \implies view-function-of-inv \Gamma$

and $v \notin fvE loc$

shows *hoare-triple-valid* $\Delta (Exists v (PointsTo loc pwrite (Evar v))) (Cwrite loc E) (PointsTo loc pwrite E)$

<proof>

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*:

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

lemma *safe-read-Some*:

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

lemma *safe-read*:

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

theorem *read-rule*:

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

4.4.6 Share

lemma *share-no-abort*:

assumes *no-abort* (*Some* Γ) $C \ s \ (h :: ('i, 'a) \text{ heap})$
and *Some* ($h' :: ('i, 'a) \text{ heap}$) = *Some* $h \oplus \text{Some } hj$
and *sat-inv* $s \ hj \ \Gamma$
and *get-gs* $h = \text{Some} \text{ (pwrite, sargs)}$
and $\bigwedge k. \text{get-gu } h \ k = \text{Some} \text{ (uargs } k)$
and *reachable-value* (*saction* Γ) (*uaction* Γ) $v0 \ \text{sargs} \ \text{uargs} \text{ (view } \Gamma \text{ (normalize (get-fh } hj)))}$
and *view-function-of-inv* Γ
shows *no-abort* *None* $C \ s \text{ (remove-guards } h')$
 $\langle \text{proof} \rangle$

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

$S\text{-after-share } S \ \Gamma \ v0 = \{ (s, \text{remove-guards } h') \mid h \ hj \ h' \ s. \text{semi-consistent } \Gamma \ v0 \ h' \wedge \text{Some } h' = \text{Some } h \oplus \text{Some } hj \wedge (s, h) \in S \wedge \text{sat-inv } s \ hj \ \Gamma \}$

lemma *share-lemma*:

assumes $\text{safe } n \text{ (Some } \Gamma) \ C \ (s, h :: ('i, 'a) \text{ heap}) \ S$
and *Some* ($h' :: ('i, 'a) \text{ heap}$) = *Some* $h \oplus \text{Some } hj$
and *sat-inv* $s \ hj \ \Gamma$
and *semi-consistent* $\Gamma \ v0 \ h'$

and *view-function-of-inv* Γ
shows *safe n* (*None* :: ('i, 'a, nat) cont) C (*s*, *remove-guards h'*) (*S-after-share*
 $S \Gamma v0$)
 <proof>

definition *no-need-guards where*

no-need-guards $A \iff (\forall s1 h1 s2 h2. (s1, h1), (s2, h2) \models A \implies (s1, \text{remove-guards } h1), (s2, \text{remove-guards } h2) \models A)$

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

assumes $\neg \text{no-guard } h$
and $C = \text{Cskip} \implies (s, h) \in S$
shows *safe n* (*None* :: ('i, 'a, nat) cont) C (*s*, *h*) S
 <proof>

theorem *share-rule:*

fixes $\Gamma :: ('i, 'a, nat)$ *single-context*
assumes $\Gamma = \langle \text{view} = f, \text{abstract-view} = \alpha, \text{saction} = \text{sact}, \text{uaction} = \text{uact}, \text{invariant} = J \rangle$
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 $\bigwedge n. \text{ safe } n \text{ (None :: ('i, 'a, nat) cont) } C1 \text{ (s1, h) } S$
and $H = \text{denormalize } H1$
and $\text{Some } H = \text{Some } h \oplus \text{Some } hf$
and $\text{full-ownership (get-fh } H) \wedge \text{no-guard } H$
shows $\neg \text{aborts } C2 \ \sigma 2 \wedge (C2 = \text{Cskip} \longrightarrow$
 $(\exists h1 \ H'. \text{Some } H' = \text{Some } h1 \oplus \text{Some } hf \wedge H2 = \text{normalize (get-fh (H'))} \wedge$
 $\text{no-guard } H' \wedge \text{full-ownership (get-fh } H') \wedge (s2, h1) \in S))$
 $\langle \text{proof} \rangle$

theorem *atomic-rule-unique:*

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

fixes $\text{map-to-list} :: \text{nat} \Rightarrow 'a \text{ list}$

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

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

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

and $\text{precise } J \wedge \text{unary } J$

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

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

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

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 $\text{no-guard-assertion } P$

and $\text{no-guard-assertion } Q$

shows $\text{hoare-triple-valid (Some } \Gamma) \text{ (Star } P \text{ (UniqueGuard index (\lambda s. \text{map-to-list}$
 $(s \ l)))) \text{ (Catomic } C)$

$(\text{Star } Q \text{ (UniqueGuard index (\lambda s. \text{map-to-arg (s } \text{uarg}) \#$
 $\text{map-to-list (s } l))))$
 $\langle \text{proof} \rangle$

theorem *atomic-rule-shared:*

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

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

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

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

and $\text{hoare-triple-valid (None :: ('i, 'a, nat) cont) (Star } P \text{ (View } f \ J \ (\lambda s. s$

$x))) C$
 $(Star Q (View f J (\lambda s. sact (s x) (map-to-arg (s sarg))))))$
and $precise J \wedge unary J$
and $view-function-of-inv \Gamma$
and $x \notin fvC C \cup fvA P \cup fvA Q \cup fvA J$

and $sarg \notin fvC C$
and $ms \notin fvC C$

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

and $no-guard-assertion P$
and $no-guard-assertion Q$

shows $hoare-triple-valid (Some \Gamma) (Star P (SharedGuard \pi (\lambda s. map-to-multiset (s ms)))) (Catomic C)$
 $(Star Q (SharedGuard \pi (\lambda s. \{\# map-to-arg (s sarg) \# \} + map-to-multiset (s ms))))$
 $\langle proof \rangle$

4.4.8 Parallel

lemma *par-cases*:

assumes $red (Cpar C1 C2) \sigma C' \sigma'$
and $\bigwedge C1'. C' = Cpar C1' C2 \wedge red C1 \sigma C1' \sigma' \implies P$
and $\bigwedge C2'. C' = Cpar C1 C2' \wedge red C2 \sigma C2' \sigma' \implies P$
and $C1 = Cskip \wedge C2 = Cskip \wedge C' = Cskip \wedge \sigma = \sigma' \implies P$
shows P
 $\langle proof \rangle$

lemma *no-abort-par*:

assumes $no-abort \Gamma C1 s h$
and $no-abort \Gamma C2 s h$
shows $no-abort \Gamma (Cpar C1 C2) s h$
 $\langle proof \rangle$

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 $upper-fvs S2 vars2$

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 Γ) $C1$ (s , $h1$) $S1$
and *safe* n (Some Γ) $C2$ (s , $h2$) $S2$
and *Some* $h = \text{Some } h1 \oplus \text{Some } h2$

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

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

and *disjoint* (*fvA* (*invariant* Γ)) (*wrC* $C2$)
and *disjoint* (*fvA* (*invariant* Γ)) (*wrC* $C1$)

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

lemma *parallel-comp*:

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

assumes *safe* n Δ $C1$ (s , $h1$) $S1$
and *safe* n Δ $C2$ (s , $h2$) $S2$
and *Some* $h = \text{Some } h1 \oplus \text{Some } h2$
and *disjoint* (*fvC* $C1 \cup \text{vars1}$) (*wrC* $C2$)
and *disjoint* (*fvC* $C2 \cup \text{vars2}$) (*wrC* $C1$)
and *upper-fvs* $S1$ *vars1*
and *upper-fvs* $S2$ *vars2*

and $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies \text{disjoint} (\text{fvA } (\text{invariant } \Gamma)) (\text{wrC } C2)$
and $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies \text{disjoint} (\text{fvA } (\text{invariant } \Gamma)) (\text{wrC } C1)$

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

theorem *rule-par*:

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

assumes *hoare-triple-valid* Δ $P1$ $C1$ $Q1$
and *hoare-triple-valid* Δ $P2$ $C2$ $Q2$
and *disjoint* (*fvA* $P1 \cup \text{fvC } C1 \cup \text{fvA } Q1$) (*wrC* $C2$)
and *disjoint* (*fvA* $P2 \cup \text{fvC } C2 \cup \text{fvA } Q2$) (*wrC* $C1$)

and $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies \text{disjoint} (\text{fvA } (\text{invariant } \Gamma)) (\text{wrC } C2)$
and $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies \text{disjoint} (\text{fvA } (\text{invariant } \Gamma)) (\text{wrC } C1)$

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 $C' = C1 \implies s = s' \wedge h = h' \implies bdenot\ b\ s \implies P$
and $C' = C2 \implies s = s' \wedge h = h' \implies \neg bdenot\ b\ s \implies P$
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* Δ (*And* P (*Bool* b)) $C1\ Q$
and *hoare-triple-valid* Δ (*And* P (*Bool* (*Bnot* b))) $C2\ Q$
shows *hoare-triple-valid* Δ (*And* P (*Low* b)) (*Cif* $b\ C1\ C2$) Q
(*proof*)

theorem *if2-rule*:

fixes $\Delta :: ('i, 'a, nat)\ cont$
assumes *hoare-triple-valid* Δ (*And* P (*Bool* b)) $C1\ Q$
and *hoare-triple-valid* Δ (*And* P (*Bool* (*Bnot* b))) $C2\ Q$
and *unary* Q

shows *hoare-triple-valid* $\Delta P (Cif\ b\ C1\ C2) Q$
 $\langle proof \rangle$

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* $C1\ \sigma$
 $\langle proof \rangle$

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

lemma *safe-seq-Some*:
assumes *safe* $n (Some\ \Gamma) C1 (s, h) S1$
and $\bigwedge m\ s'\ h'. m \leq n \wedge (s', h') \in S1 \implies safe\ m (Some\ \Gamma) C2 (s', h') S2$
shows *safe* $n (Some\ \Gamma) (Cseq\ C1\ C2) (s, h) S2$
 $\langle proof \rangle$

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

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

4.4.11 Frame rule

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

lemma *safe-frame-Some*:

assumes *safe n* (Some Γ) C (s, h) S
and Some $H = \text{Some } h \oplus \text{Some } hf0$
shows *safe n* (Some Γ) C (s, H) (*add-states* $S \{(s'', hf0) | s''. \text{ agrees } (- \text{ wrC } C) s s''\}$)
<proof>

lemma *safe-frame*:

fixes $\Delta :: ('i, 'a, \text{ nat}) \text{ cont}$
assumes *safe n* Δ C (s, h) S
and Some $H = \text{Some } h \oplus \text{Some } hf0$
shows *safe n* Δ C (s, H) (*add-states* $S \{(s'', hf0) | s''. \text{ agrees } (- \text{ wrC } C) s s''\}$)
<proof>

theorem *frame-rule*:

fixes $\Delta :: ('i, 'a, \text{ nat}) \text{ cont}$
assumes *hoare-triple-valid* Δ P C Q
and *disjoint* ($\text{fv} A$ R) ($\text{wrC } C$)
and *precise* $P \vee \text{ precise } R$
shows *hoare-triple-valid* Δ (*Star* P R) C (*Star* Q R)
<proof>

4.4.12 Consequence

theorem *consequence-rule*:

fixes $\Delta :: ('i, 'a, \text{ nat}) \text{ cont}$
assumes *hoare-triple-valid* Δ P' C Q'
and *entails* P P'
and *entails* Q' Q
shows *hoare-triple-valid* Δ P C Q
<proof>

4.4.13 Existential

theorem *existential-rule*:

fixes $\Delta :: ('i, 'a, \text{ nat}) \text{ cont}$
assumes *hoare-triple-valid* Δ P C Q
and $x \notin \text{fv} C$
and $\bigwedge \Gamma. \Delta = \text{Some } \Gamma \implies x \notin \text{fv} A$ (*invariant* Γ)
and *unambiguous* P x
shows *hoare-triple-valid* Δ (*Exists* x P) C (*Exists* x Q)
<proof>

4.4.14 While loops

inductive *leads-to-loop where*

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

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

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

definition *trans- Σ* **where**

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

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

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

lemma *safe-while-None*:

assumes $\bigwedge \sigma m. \sigma, \sigma \models \text{And } I (\text{Bool } b) \implies \text{safe } n (\text{None} :: ('i, 'a, \text{nat}) \text{cont}) C \sigma (\Sigma \sigma)$

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 *leads-to-loop* $b I \Sigma \sigma (s, h)$

shows $\text{safe } n (\text{None} :: ('i, 'a, \text{nat}) \text{cont}) (C\text{while } b C) (s, h) (\text{trans-}\Sigma b I \Sigma \sigma)$

<proof>

lemma *safe-while-Some*:

assumes $\bigwedge \sigma m. \sigma, \sigma \models \text{And } I (\text{Bool } b) \implies \text{safe } n (\text{Some } \Gamma) C \sigma (\Sigma \sigma)$

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 *leads-to-loop* $b I \Sigma \sigma (s, h)$

shows $\text{safe } n (\text{Some } \Gamma) (C\text{while } b C) (s, h) (\text{trans-}\Sigma b I \Sigma \sigma)$

<proof>

lemma *safe-while*:

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

assumes $\bigwedge \sigma m. \sigma, \sigma \models \text{And } I (\text{Bool } b) \implies \text{safe } n \Delta C \sigma (\Sigma \sigma)$

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 *leads-to-loop* $b I \Sigma \sigma (s, h)$

shows $\text{safe } n \Delta (C\text{while } b C) (s, h) (\text{trans-}\Sigma b I \Sigma \sigma)$

<proof>

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

assumes *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 $\sigma', \sigma' \models I$

<proof>

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* $\Delta I (C\text{while } b C) (\text{And } I (\text{Bool } (\text{Bnot } b)))$

<proof>

fun *iterate-sigma* :: *nat* \Rightarrow *bexp* \Rightarrow ('i, 'a, *nat*) *assertion* \Rightarrow ((*store* \times ('i, 'a) *heap*) \Rightarrow (*store* \times ('i, 'a) *heap*) *set*) \Rightarrow (*store* \times ('i, 'a) *heap*) \Rightarrow (*store* \times ('i, 'a) *heap*) \Rightarrow (*store* \times ('i, 'a) *heap*) *set*

where

iterate-sigma 0 *b I* Σ $\sigma = \{\sigma\}$

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

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

assumes *leads-to-loop* *b I* Σ σ σ'

shows $\sigma' \in (\bigcup n. \text{iterate-sigma } n \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma)$

<proof>

lemma *trans-included*:

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

<proof>

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

assumes $\bigwedge \sigma \sigma'. \sigma, \sigma' \models (\text{And } I \text{ (Bool } b)) \Longrightarrow \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') (\text{And } I \text{ (Low } b))$

and $\sigma 1, \sigma 2 \models I$

and $\text{bdenot } b \text{ (fst } \sigma 1) = \text{bdenot } b \text{ (fst } \sigma 2)$

shows $\text{pair-sat } (\text{iterate-sigma } n \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma 1) (\text{iterate-sigma } n \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma 2) (\text{And } I \text{ (Low } b))$

<proof>

lemma *iterate-empty-later-empty*:

assumes *iterate-sigma* *n b I* Σ $\sigma = \{\}$

and $m \geq n$

shows *iterate-sigma* *m b I* Σ $\sigma = \{\}$

<proof>

lemma *all-same*:

assumes $\bigwedge \sigma \sigma'. \sigma, \sigma' \models (\text{And } I \text{ (Bool } b)) \Longrightarrow \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') (\text{And } I \text{ (Low } b))$

and $\sigma 1, \sigma 2 \models I$

and $\text{bdenot } b \text{ (fst } \sigma 1) = \text{bdenot } b \text{ (fst } \sigma 2)$

and $x 1 \in \text{iterate-sigma } n \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma 1$

and $x 2 \in \text{iterate-sigma } n \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma 2$

shows $\text{bdenot } b \text{ (fst } x 1) = \text{bdenot } b \text{ (fst } x 2)$

<proof>

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

assumes $\bigwedge \sigma \sigma'. \sigma, \sigma' \models (\text{And } I \text{ (Bool } b)) \Longrightarrow \text{pair-sat } (\Sigma \sigma) (\Sigma \sigma') (\text{And } I \text{ (Low } b))$

and $\sigma, \sigma \models I$
and $\text{Set.filter } (\lambda\sigma. \neg \text{bdenot } b \text{ (fst } \sigma)) \text{ (iterate-sigma } n1 \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma) \neq \{\}$
and $\text{Set.filter } (\lambda\sigma. \neg \text{bdenot } b \text{ (fst } \sigma)) \text{ (iterate-sigma } n2 \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma) \neq \{\}$
shows $n1 = n2$
 $\langle \text{proof} \rangle$

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 $\text{Set.filter } (\lambda\sigma. \neg \text{bdenot } b \text{ (fst } \sigma)) \text{ (iterate-sigma } k \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma) \neq \{\}$
shows $\text{Set.filter } (\lambda\sigma. \neg \text{bdenot } b \text{ (fst } \sigma)) (\bigcup n. \text{iterate-sigma } n \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma) = \text{Set.filter } (\lambda\sigma. \neg \text{bdenot } b \text{ (fst } \sigma)) \text{ (iterate-sigma } k \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma)$
 $\langle \text{proof} \rangle$

definition *not-set where*

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

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 $\sigma1, \sigma2 \models I$
and $\text{bdenot } b \text{ (fst } \sigma1) = \text{bdenot } b \text{ (fst } \sigma2)$
and $\text{Set.filter } (\lambda\sigma. \neg \text{bdenot } b \text{ (fst } \sigma)) (\bigcup n. \text{iterate-sigma } n \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma1) \neq \{\}$
and $\text{Set.filter } (\lambda\sigma. \neg \text{bdenot } b \text{ (fst } \sigma)) (\bigcup n. \text{iterate-sigma } n \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma2) \neq \{\}$
shows $\exists k. \text{not-set } b \text{ } (\bigcup n. \text{iterate-sigma } n \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma1) = \text{not-set } b \text{ } (\text{iterate-sigma } k \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma1) \wedge \text{not-set } b \text{ } (\bigcup n. \text{iterate-sigma } n \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma2) = \text{not-set } b \text{ } (\text{iterate-sigma } k \text{ } b \text{ } I \text{ } \Sigma \text{ } \sigma2)$
 $\langle \text{proof} \rangle$

theorem *while-rule1*:

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

lemma *entails-smallerI*:

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

corollary *while-rule*:

fixes $\Delta :: ('i, 'a, \text{nat}) \text{ cont}$
assumes $\text{entails } P \text{ (Star } P' \text{ } R)$
and $\text{unary } P'$

and $fvA\ R \cap wrC\ C = \{\}$
and *hoare-triple-valid* $\Delta\ (And\ P'\ (Bool\ e))\ C\ P'$
and *hoare-triple-valid* $\Delta\ (And\ P\ (Bool\ (Band\ e\ e')))\ C\ (And\ P\ (Low\ (Band\ e\ e')))$
and *precise* $P' \vee$ *precise* R
shows *hoare-triple-valid* $\Delta\ (And\ P\ (Low\ (Band\ e\ e')))\ (Cseq\ (Cwhile\ (Band\ e\ e')\ C)\ (Cwhile\ e\ C))\ (And\ (Star\ P'\ R)\ (Bool\ (Bnot\ e)))$
 \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

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 $\bigwedge \sigma1'\ \sigma2'.\ red\text{-}rtrans\ C\ (s1,\ normalize\ (get\text{-}fh\ H1))\ Cskip\ \sigma1' \implies red\text{-}rtrans\ C\ (s2,\ normalize\ (get\text{-}fh\ H2))\ Cskip\ \sigma2'$
 $\implies (\exists h1'\ h2'\ H1'\ H2'.\ no\text{-}guard\ H1' \wedge full\text{-}ownership\ (get\text{-}fh\ H1') \wedge snd\ \sigma1' = normalize\ (get\text{-}fh\ H1') \wedge Some\ H1' = Some\ h1' \oplus Some\ hf1 \wedge no\text{-}guard\ H2' \wedge full\text{-}ownership\ (get\text{-}fh\ H2') \wedge snd\ \sigma2' = normalize\ (get\text{-}fh\ H2') \wedge Some\ H2' = Some\ h2' \oplus Some\ hf2 \wedge (fst\ \sigma1',\ h1'), (fst\ \sigma2',\ h2') \models Q)$
 \langle *proof* \rangle

lemma *neutral-add*:

$Some\ h = Some\ h \oplus Some\ (Map.empty,\ None,\ (\lambda\cdot.\ None))$
 \langle *proof* \rangle

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  $\bigwedge \sigma' C'. \text{red-rtrans } C (s1, \text{normalize (get-fh H1)}) C' \sigma' \implies \neg \text{aborts } C'$ 
 $\sigma'$ 
and  $\bigwedge \sigma' C'. \text{red-rtrans } C (s2, \text{normalize (get-fh H2)}) C' \sigma' \implies \neg \text{aborts } C'$ 
 $\sigma'$ 
and  $\bigwedge \sigma 1' \sigma 2'. \text{red-rtrans } C (s1, \text{normalize (get-fh H1)}) C \text{skip } \sigma 1'$ 
 $\implies \text{red-rtrans } C (s2, \text{normalize (get-fh H2)}) C \text{skip } \sigma 2'$ 
 $\implies (\exists H1' H2'. \text{no-guard } H1' \wedge \text{full-ownership (get-fh H1')} \wedge \text{snd } \sigma 1' = \text{normalize (get-fh H1')})$ 
 $\wedge \text{no-guard } H2' \wedge \text{full-ownership (get-fh H2')} \wedge \text{snd } \sigma 2' = \text{normalize (get-fh H2')}$ 
 $\wedge (\text{fst } \sigma 1', H1'), (\text{fst } \sigma 2', H2') \models Q$ 
 $\langle \text{proof} \rangle$ 

end
theory NonInterference
imports Soundness
begin

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  $\in$  set Out
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 s1' x = s2' x
 $\langle \text{proof} \rangle$ 

```

definition *heapify where*

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

lemma *heapify-properties:*

full-ownership $(\text{get-fh } (\text{heapify } h))$

no-guard $(\text{heapify } h)$

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 *red-rtrans* $C (s1, h1) Cskip (s1', h1')$

and *red-rtrans* $C (s2, h2) Cskip (s2', h2')$

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.