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

May 26, 2024

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 CommCSL [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

1.1 Partial Heaps

theory *PartialMap*
imports *Main*
begin

type-synonym $('a, 'b)\ map = 'a \rightharpoonup 'b$

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 $\bigwedge x\ a\ b.\ h1\ x = Some\ a \wedge h2\ x = Some\ b \Longrightarrow f\ a\ b$
shows *compatible-maps f h1 h2*
⟨*proof*⟩

definition *map-included* :: $('a, 'b)\ map \Rightarrow ('a, 'b)\ map \Rightarrow bool$ **where**
map-included h1 h2 $\longleftrightarrow$ ($\forall x.\ h1\ x \neq None \longrightarrow h1\ x = h2\ x$)

lemma *map-includedI*:
assumes $\bigwedge x\ r.\ h1\ x = Some\ r \Longrightarrow h2\ x = Some\ r$
shows *map-included h1 h2*
⟨*proof*⟩

lemma *compatible-maps-empty*:
compatible-maps f h (*Map.empty*)
⟨*proof*⟩

lemma *compatible-maps-comm*:
compatible-maps (=) *h1 h2* $\longleftrightarrow$ *compatible-maps* (=) *h2 h1*
⟨*proof*⟩

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 *pwrite* :: *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* ⇒ *prat* ⇒ *prat* **is** (/) ⟨*proof*⟩

lift-definition *pmin* :: *prat* ⇒ *prat* ⇒ *prat* **is** (*min*) ⟨*proof*⟩
lift-definition *pmax* :: *prat* ⇒ *prat* ⇒ *prat* **is** (*max*) ⟨*proof*⟩

lemma *pmin-comm*:
pmin a b = *pmin b a*
⟨*proof*⟩

lemma *pmin-greater*:
pgte a (*pmin a b*)
⟨*proof*⟩

lemma *pmin-is*:
assumes *pgte a b*
shows *pmin a b* = *b*
⟨*proof*⟩

lemma *pmax-comm*:
pmax a b = *pmax b a*
⟨*proof*⟩

lemma *pmax-smaller*:
pgte (*pmax a b*) *a*
⟨*proof*⟩

lemma *pmax-is*:
assumes *pgte a b*
shows *pmax a b* = *a*
⟨*proof*⟩

lemma *pmax-is-smaller*:
assumes *pgte x a*
and *pgte x b*
shows *pgte x* (*pmax a b*)
⟨*proof*⟩

lemma *half-between-0-1*:
pgt pwrite half
⟨*proof*⟩

lemma *pgt-implies-pgte*:
assumes *pgt a b*
shows *pgte a b*
⟨*proof*⟩

lemma *half-plus-half*:
padd half half = pwrite
⟨*proof*⟩

lemma *padd-comm*:
padd a b = padd b a
⟨*proof*⟩

lemma *padd-asso*:
padd (*padd a b*) *c* = *padd a* (*padd b c*)
⟨*proof*⟩

lemma *pgte-antisym*:
assumes *pgte a b*
and *pgte b a*
shows *a = b*
⟨*proof*⟩

lemma *sum-larger*:
pgt (*padd a b*) *a*
⟨*proof*⟩

lemma *greater-sum-both*:
assumes *pgte a* (*padd b c*)
shows ∃ *a1 a2*. *a = padd a1 a2* ∧ *pgte a1 b* ∧ *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*:
¬ *pgte a b* ⟷ *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* $= 'l \rightharpoonup prat \times 'v$

definition *compatible-fractions* :: $('l, 'v)$ *fract-heap* $\Rightarrow$ $('l, 'v)$ *fract-heap* $\Rightarrow$ *bool*
where
compatible-fractions h h' $\longleftrightarrow$
$(\forall l\ p\ p'.\ h\ l = Some\ p \wedge h'\ l = Some\ p' \longrightarrow pgte\ pwrite\ (padd\ (fst\ p)\ (fst\ p')))$

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

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 $\bigwedge l\ p\ p'.\ h\ l = Some\ p \wedge h'\ l = Some\ p' \Longrightarrow pgte\ pwrite\ (padd\ (fst\ p)\ (fst\ p'))$
and $\bigwedge l\ p\ p'.\ h\ l = Some\ p \wedge h'\ l = Some\ p' \Longrightarrow snd\ p = snd\ p'$
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 pwrite* $(padd\ (fst\ p)\ (fst\ p'))$
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* ⇒ ('*l*, '*v*) *fract-heap* ⇒ ('*l*, '*v*) *fract-heap*
where
add-fh h h' l = *fadd-options* (*h l*) (*h' l*)

definition *full-ownership* :: ('*l*, '*v*) *fract-heap* ⇒ *bool* **where**
full-ownership h ⟷ (∀ *l p*. *h l* = *Some p* ⟶ *fst p* = *pwrite*)

lemma *full-ownershipI*:
assumes ⋀*l p*. *h l* = *Some p* ⟹ *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* ⇒ ('*l* ⇀ '*v*) **where**
normalize h l = *apply-opt snd* (*h l*)

lemma *normalize-eq*:
normalize h l = *None* ⟷ *h l* = *None*
normalize h l = *Some v* ⟷ (∃ *p*. *h l* = *Some* (*p*, *v*)) (**is** *?A* ⟷ *?B*)
⟨*proof*⟩

definition *fpdom* **where**
fpdom h = {*x*. ∃ *v*. *h x* = *Some* (*pwrite*, *v*)}

lemma *compatible-then-dom-disjoint*:
assumes *compatible-fract-heaps h1 h2*
shows *dom h1* ∩ *fpdom h2* = {}
and *dom h2* ∩ *fpdom h1* = {}
⟨*proof*⟩

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

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

lemma *add-fh-update*:
assumes *b x* = *None*

shows *add-fh* $(a(x \mapsto p))\ b = (\mathit{add\text{-}fh}\ a\ b)(x \mapsto p)$
⟨*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-heap* = (*loc*, *val*) *fract-heap*
type-synonym *'a gs-heap* = (*prat* × *'a multiset*) *option*
type-synonym (*'i*, *'a*) *gu-heap* = *'i* ⇒ *'a list option*

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

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

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

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

definition *compatible* :: (*'i*, *'a*) *heap* ⇒ (*'i*, *'a*) *heap* ⇒ *bool* (**infixl** *##* *60*) **where**
h ## h′ ⟷ *compatible-fract-heaps* (*get-fh h*) (*get-fh h′*) ∧ (∀ *k*. *get-gu h k* = *None* ∨ *get-gu h′ k* = *None*)
∧ (∀ *p p′*. *get-gs h* = *Some p* ∧ *get-gs h′* = *Some p′* ⟶ *pgte pwrite* (*padd* (*fst p*) (*fst p′*)))

lemma *compatibleI*:
assumes *compatible-fract-heaps* (*get-fh h*) (*get-fh h′*)
and ⋀*k*. *get-gu h k* = *None* ∨ *get-gu h′ k* = *None*
and ⋀*p p′*. *get-gs h* = *Some p* ∧ *get-gs h′* = *Some p′* ⟹ *pgte pwrite* (*padd* (*fst p*) (*fst p′*))
shows *h ## h′*
⟨*proof*⟩

fun *add-gu-single* **where**

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

definition *add-gu* **where**
add-gu u1 u2 k = add-gu-single (u1 k) (u2 k)

lemma *comp-add-gu-comm*:
assumes $\bigwedge k.\ h\ k = None \vee h'\ k = None$
shows *add-gu h h' = add-gu h' h*
⟨*proof*⟩

fun *add-gs* :: (*prat* × *'a multiset*) *option* ⇒ (*prat* × *'a multiset*) *option* ⇒ (*prat* × *'a multiset*) *option*
where
add-gs None x = x
| *add-gs x None = x*
| *add-gs (Some p) (Some p') = Some (padd (fst p) (fst p'), snd p + snd p')*

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 ⟷ b ## a
⟨*proof*⟩

lemma *heap-ext*:
assumes *get-fh a = get-fh b*
and *get-gs a = get-gs b*
and *get-gu a = get-gu b*
shows *a = b*
⟨*proof*⟩

lemma *plus-comm*:
a ⊕ b = b ⊕ a
⟨*proof*⟩

lemma *asso2*:
assumes *Some a ⊕ Some b = Some ab*
and *¬ b ## c*
shows *¬ ab ## c*
⟨*proof*⟩

lemma *plus-extract-point-fh*:
assumes *Some x = Some a ⊕ Some b*
and *get-fh a l = Some pa*
and *get-fh b l = Some pb*
shows *snd pa = snd pb ∧ pgte pwrite (padd (fst pa) (fst pb)) ∧ get-fh x l = Some (padd (fst pa) (fst pb), snd pa)*
⟨*proof*⟩

lemma *asso1*:
assumes *Some a ⊕ Some b = Some ab*
and *Some b ⊕ Some c = Some bc*
shows *Some ab ⊕ Some c = Some a ⊕ Some bc*
⟨*proof*⟩

lemma *simpler-asso*:
(Some a ⊕ Some b) ⊕ Some c = Some a ⊕ (Some b ⊕ Some c)
⟨*proof*⟩

lemma *plus-asso*:
(a ⊕ b) ⊕ c = a ⊕ (b ⊕ c)
⟨*proof*⟩

definition *larger* :: *('i, 'a) heap ⇒ ('i, 'a) heap ⇒ bool* (**infixl** *≽ 55*) **where**
a ≽ b ⟷ (∃ c. Some a = Some b ⊕ 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 $Some\ b = Some\ c \oplus Some\ d$
shows $a \#\# c$
⟨*proof*⟩

lemma *full-sguard-sum-same*:
assumes $get\text{-}gs\ a = Some\ (pwrite, sargs)$
and $Some\ h = Some\ a \oplus Some\ b$
shows $get\text{-}gs\ h = Some\ (pwrite, sargs)$
⟨*proof*⟩

lemma *full-uguard-sum-same*:
assumes $get\text{-}gu\ a\ k = Some\ uargs$
and $Some\ h = Some\ a \oplus Some\ b$
shows $get\text{-}gu\ h\ k = Some\ uargs$
⟨*proof*⟩

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

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

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

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

lemma *larger3*:
assumes *Some x* = *Some a* ⊕ *Some b* ⊕ *Some c*
shows *x* ⪰ *b*
⟨*proof*⟩

lemma *add-get-fh*:
assumes *Some x* = *Some a* ⊕ *Some b*
shows *get-fh x* = *add-fh* (*get-fh a*) (*get-fh b*)
⟨*proof*⟩

lemma *sum-gs-one-none*:
assumes *Some x* = *Some a* ⊕ *Some b*
and *get-gs b* = *None*
shows *get-gs x* = *get-gs a*
⟨*proof*⟩

lemma *sum-gs-one-some*:
assumes *Some x* = *Some a* ⊕ *Some b*
and *get-gs a* = *Some* (*pa*, *ma*)
and *get-gs b* = *Some* (*pb*, *mb*)
shows *get-gs x* = *Some* (*padd pa pb*, *ma* + *mb*)
⟨*proof*⟩

definition *empty-heap* :: ('*i*, '*a*) *heap* **where**
empty-heap = (*Map.empty*, *None*, λ*k*. *None*)

lemma *dom-normalize*:
dom h = *dom* (*normalize h*)
⟨*proof*⟩

lemma *sum-second-none-get-fh*:
assumes *Some x* = *Some a* ⊕ *Some b*
and *get-fh b l* = *None*
shows *get-fh x l* = *get-fh a l*
⟨*proof*⟩

lemma *sum-first-none-get-fh*:
assumes *Some x* = *Some a* ⊕ *Some b*
and *get-fh a l* = *None*
shows *get-fh x l* = *get-fh b l*
⟨*proof*⟩

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* ⇒ *store* ⇒ *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* ⇒ *store* ⇒ *bool*
where
  bdenot (*Beq e1 e2*) *s* = (*edenot e1 s* = *edenot e2 s*)
| *bdenot* (*Band b1 b2*) *s* = (*bdenot b1 s* ∧ *bdenot b2 s*)
| *bdenot* (*Bnot b*) *s* = (¬ *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* ⇒ *state* ⇒ *cmd* ⇒ *state* ⇒ *bool*
and *red-rtrans* :: *cmd* ⇒ *state* ⇒ *cmd* ⇒ *state* ⇒ *bool*
where
  red-Seq1[*intro*]: *red* (*Cseq Cskip C*) *σ C σ*
| *red-Seq2*[*elim*]: *red C1 σ C1′ σ′* ⟹ *red* (*Cseq C1 C2*) *σ* (*Cseq C1′ C2*) *σ′*
| *red-If1*[*intro*]: *bdenot B* (*fst σ*) ⟹ *red* (*Cif B C1 C2*) *σ C1 σ*
| *red-If2*[*intro*]: ¬ *bdenot B* (*fst σ*) ⟹ *red* (*Cif B C1 C2*) *σ C2 σ*
| *red-Atomic*[*intro*]: *red-rtrans C σ Cskip σ′* ⟹ *red* (*Catomic C*) *σ Cskip σ′*
| *red-Par1*[*elim*]: *red C1 σ C1′ σ′* ⟹ *red* (*Cpar C1 C2*) *σ* (*Cpar C1′ C2*) *σ′*
| *red-Par2*[*elim*]: *red C2 σ C2′ σ′* ⟹ *red* (*Cpar C1 C2*) *σ* (*Cpar C1 C2′*) *σ′*
| *red-Par3*[*intro*]: *red* (*Cpar Cskip Cskip*) *σ* (*Cskip*) *σ*
| *red-Loop*[*intro*]: *red* (*Cwhile B C*) *σ* (*Cif B* (*Cseq C* (*Cwhile B C*)) *Cskip*) *σ*
| *red-Assign*[*intro*]:⟦ *σ* = (*s*,*h*); *σ′* = (*s*(*x* := *edenot E s*), *h*) ⟧ ⟹ *red* (*Cassign x E*) *σ Cskip σ′*
| *red-Read*[*intro*]: ⟦ *σ* = (*s*,*h*); *h*(*edenot E s*) = *Some v*; *σ′* = (*s*(*x* := *v*), *h*) ⟧ ⟹ *red* (*Cread x E*) *σ Cskip σ′*
| *red-Write*[*intro*]: ⟦ *σ* = (*s*,*h*); *σ′* = (*s*, *h*(*edenot E s* ↦ *edenot E′ s*)) ⟧ ⟹ *red* (*Cwrite E E′*) *σ Cskip σ′*
| *red-Alloc*[*intro*]: ⟦ *σ* = (*s*,*h*); *v* ∉ *dom h*; *σ′* = (*s*(*x* := *v*), *h*(*v* ↦ *edenot E s*)) ⟧ ⟹ *red* (*Calloc x E*) *σ Cskip σ′*
| *red-Free*[*intro*]: ⟦ *σ* = (*s*,*h*); *σ′* = (*s*, *h*(*edenot E s* := *None*)) ⟧ ⟹ *red* (*Cdispose E*) *σ Cskip σ′*

| *NoStep*: *red-rtrans C σ C σ*
| *OneMoreStep*: ⟦ *red C σ C′ σ′* ; *red-rtrans C′ σ′ C″ σ″* ⟧ ⟹ *red-rtrans C σ C″ σ″*

inductive-cases *red-par-cases*: *red* (*Cpar C1 C2*) *σ C′ σ′*
inductive-cases *red-atomic-cases*: *red* (*Catomic C*) *σ C′ σ′*

2.1.3 Abort semantics

inductive
  aborts :: *cmd* ⇒ *state* ⇒ *bool*
where
  aborts-Seq[*intro*]: *aborts C1 σ* ⟹ *aborts* (*Cseq C1 C2*) *σ*
| *aborts-Atomic*[*intro*]: ⟦ *red-rtrans C σ C′ σ′* ; *aborts C′ σ′* ⟧ ⟹ *aborts* (*Catomic C*) *σ*
| *aborts-Par1*[*intro*]: *aborts C1 σ* ⟹ *aborts* (*Cpar C1 C2*) *σ*
| *aborts-Par2*[*intro*]: *aborts C2 σ* ⟹ *aborts* (*Cpar C1 C2*) *σ*
| *aborts-Read*[*intro*]: *edenot E* (*fst σ*) ∉ *dom* (*snd σ*) ⟹ *aborts* (*Cread x E*) *σ*
| *aborts-Write*[*intro*]: *edenot E* (*fst σ*) ∉ *dom* (*snd σ*) ⟹ *aborts* (*Cwrite E E′*) *σ*
| *aborts-Free*[*intro*]: *edenot E* (*fst σ*) ∉ *dom* (*snd σ*) ⟹ *aborts* (*Cdispose E*) *σ*

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* ⇒ *var set*
where
  fvE (*Evar v*) = {*v*}
| *fvE* (*Enum n*) = {}
| *fvE* (*Eplus e1 e2*) = (*fvE e1* ∪ *fvE e2*)

primrec
  fvB :: *bexp* ⇒ *var set*
where
  fvB (*Beq e1 e2*) = (*fvE e1* ∪ *fvE e2*)
| *fvB* (*Band b1 b2*) = (*fvB b1* ∪ *fvB b2*)
| *fvB* (*Bnot b*) = (*fvB b*)
| *fvB Btrue* = {}

primrec
  fvC :: *cmd* ⇒ *var set*
where
  fvC (*Cskip*) = {}
| *fvC* (*Cassign v E*) = ({*v*} ∪ *fvE E*)
| *fvC* (*Cread v E*) = ({*v*} ∪ *fvE E*)
| *fvC* (*Cwrite E1 E2*) = (*fvE E1* ∪ *fvE E2*)
| *fvC* (*Calloc v E*) = ({*v*} ∪ *fvE E*)
| *fvC* (*Cdispose E*) = (*fvE E*)
| *fvC* (*Cseq C1 C2*) = (*fvC C1* ∪ *fvC C2*)
| *fvC* (*Cpar C1 C2*) = (*fvC C1* ∪ *fvC C2*)
| *fvC* (*Cif B C1 C2*) = (*fvB B* ∪ *fvC C1* ∪ *fvC C2*)
| *fvC* (*Cwhile B C*) = (*fvB B* ∪ *fvC C*)
| *fvC* (*Catomic C*) = (*fvC C*)

primrec
  wrC :: *cmd* ⇒ *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* ∪ *wrC C2*)
| *wrC* (*Cpar C1 C2*) = (*wrC C1* ∪ *wrC C2*)
| *wrC* (*Cif B C1 C2*) = (*wrC C1* ∪ *wrC C2*)

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

primrec
$subE :: var \Rightarrow exp \Rightarrow exp \Rightarrow exp$
where
$subE\ x\ E\ (Evar\ y) \quad = (if\ x = y\ then\ E\ else\ Evar\ y)$
$|\ subE\ x\ E\ (Enum\ n) \quad = Enum\ n$
$|\ subE\ x\ E\ (Eplus\ e1\ e2) = Eplus\ (subE\ x\ E\ e1)\ (subE\ x\ E\ e2)$

primrec
$subB :: var \Rightarrow exp \Rightarrow bexp \Rightarrow bexp$
where
$subB\ x\ E\ (Beq\ e1\ e2) \ = Beq\ (subE\ x\ E\ e1)\ (subE\ x\ E\ e2)$
$|\ subB\ x\ E\ (Band\ b1\ b2) = Band\ (subB\ x\ E\ b1)\ (subB\ x\ E\ b2)$
$|\ subB\ x\ E\ (Bnot\ b) \quad = Bnot\ (subB\ x\ E\ b)$
$|\ subB\ x\ E\ Btrue = Btrue$

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

lemma *red-properties*:
red $C\ \sigma\ C'\ \sigma' \Longrightarrow fvC\ C' \subseteq fvC\ C \wedge wrC\ C' \subseteq wrC\ C \wedge agrees\ (-\ wrC\ C)\ (fst\ \sigma')\ (fst\ \sigma)$
red-rtrans $C\ \sigma\ C'\ \sigma' \Longrightarrow fvC\ C' \subseteq fvC\ C \wedge wrC\ C' \subseteq wrC\ C \wedge agrees\ (-\ wrC\ C)\ (fst\ \sigma')\ (fst\ \sigma)$
⟨*proof*⟩

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

lemma *exp-agrees*: *agrees* $(fvE\ E)\ s\ s' \Longrightarrow edenot\ E\ s = edenot\ E\ s'$
⟨*proof*⟩

lemma *bexp-agrees*:
agrees $(fvB\ B)\ s\ s' \Longrightarrow bdenot\ B\ s = bdenot\ B\ s'$
⟨*proof*⟩

lemma *red-not-in-fv-not-touched*:
red $C\ \sigma\ C'\ \sigma' \Longrightarrow x \notin fvC\ C \Longrightarrow fst\ \sigma\ x = fst\ \sigma'\ x$
red-rtrans $C\ \sigma\ C'\ \sigma' \Longrightarrow x \notin fvC\ C \Longrightarrow fst\ \sigma\ x = fst\ \sigma'\ x$
⟨*proof*⟩

lemma *agrees-update1*:
assumes *agrees* $X\ s\ s'$
shows *agrees* $X\ (s(x := v))\ (s'(x := v))$
⟨*proof*⟩

lemma *agrees-update2*:
assumes *agrees* $X\ s\ s'$
and $x \notin X$
shows *agrees* $X\ (s(x := v))\ (s'(x := v'))$
⟨*proof*⟩

lemma *red-agrees-aux*:
red $C\ \sigma\ C'\ \sigma' \Longrightarrow (\forall s\ h.\ agrees\ X\ (fst\ \sigma)\ s \wedge snd\ \sigma = h \wedge fvC\ C \subseteq X \longrightarrow$
$(\exists s'\ h'.\ red\ C\ (s, h)\ C'\ (s', h') \wedge agrees\ X\ (fst\ \sigma')\ s' \wedge snd\ \sigma' = h'))$
red-rtrans $C\ \sigma\ C'\ \sigma' \Longrightarrow (\forall s\ h.\ agrees\ X\ (fst\ \sigma)\ s \wedge snd\ \sigma = h \wedge fvC\ C \subseteq X \longrightarrow$
$(\exists s'\ h'.\ red\text{-}rtrans\ C\ (s, h)\ C'\ (s', h') \wedge agrees\ X\ (fst\ \sigma')\ s' \wedge snd\ \sigma' = h'))$
⟨*proof*⟩

lemma *red-agrees*[*rule-format*]:
red $C\ \sigma\ C'\ \sigma' \Longrightarrow \forall X\ s.\ agrees\ X\ (fst\ \sigma)\ s \longrightarrow snd\ \sigma = h \longrightarrow fvC\ C \subseteq X \longrightarrow$
$(\exists s'\ h'.\ red\ C\ (s, h)\ C'\ (s', h') \wedge agrees\ X\ (fst\ \sigma')\ s' \wedge snd\ \sigma' = h')$
⟨*proof*⟩

lemma *aborts-agrees*:

assumes *aborts C σ*
and *agrees* (*fvC C*) (*fst σ*) *s*
and *snd σ* = *h*
shows *aborts C* (*s*, *h*)
⟨*proof*⟩

corollary *exp-agrees2*[*simp*]:
x ∉ *fvE E* ⟹ *edenot E* (*s*(*x* := *v*)) = *edenot E s*
⟨*proof*⟩

lemma *agrees-update*:
assumes *a* ∉ *S*
shows *agrees S s* (*s*(*a* := *v*))
⟨*proof*⟩

lemma *agrees-comm*:
agrees S s s′ ⟷ *agrees S s′ s*
⟨*proof*⟩

lemma *not-in-dom*:
assumes *x* ∉ *dom s*
shows *s x* = *None*
⟨*proof*⟩

lemma *agrees-minusD*:
agrees (−*X*) *x y* ⟹ *X* ∩ *Y* = {} ⟹ *agrees Y x y*
⟨*proof*⟩

end

3 CommCSL

theory *CommCSL*
imports *Lang StateModel*
begin

definition *no-guard* :: (′*i*, ′*a*) *heap* ⇒ *bool* **where**
no-guard h ⟷ *get-gs h* = *None* ∧ (∀ *k*. *get-gu h k* = *None*)

typedef ′*a precondition* = { *pre* :: (′*a* ⇒ ′*a* ⇒ *bool*) |*pre*. ∀ *a b*. *pre a b* ⟶ (*pre b a* ∧ *pre a a*) }
⟨*proof*⟩

lemma *charact-rep-prec*:
assumes *Rep-precondition pre a b*
shows *Rep-precondition pre b a* ∧ *Rep-precondition pre a a*
⟨*proof*⟩

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 ⇒ *bool* (-, - ⊨ - [*51*, *65*, *66*] *50*) **where**
  (*s*, -), (*s′*, -) ⊨ *Bool b* ⟷ *bdenot b s* ∧ *bdenot b s′*
| (-, *h*), (-, *h′*) ⊨ *Emp* ⟷ *dom* (*get-fh h*) = {} ∧ *dom* (*get-fh h′*) = {}
| *σ*, *σ′* ⊨ *And A B* ⟷ *σ*, *σ′* ⊨ *A* ∧ *σ*, *σ′* ⊨ *B*

| (*s*, *h*), (*s′*, *h′*) ⊨ *Star A B* ⟷ (∃ *h1 h2 h1′ h2′*. *Some h* = *Some h1* ⊕ *Some h2* ∧ *Some h′* = *Some h1′* ⊕ *Some h2′*
  ∧ (*s*, *h1*), (*s′*, *h1′*) ⊨ *A* ∧ (*s*, *h2*), (*s′*, *h2′*) ⊨ *B*)
| (*s*, *h*), (*s′*, *h′*) ⊨ *Low e* ⟷ *bdenot e s* = *bdenot e s′*

| (*s*, *h*), (*s′*, *h′*) ⊨ *PointsTo loc p x* ⟷ *get-fh h* (*edenot loc s*) = *Some* (*p*, *edenot x s*) ∧ *get-fh h′* (*edenot loc s′*) = *Some* (*p*, *edenot x s′*)
∧ *dom* (*get-fh h*) = {*edenot loc s*} ∧ *dom* (*get-fh h′*) = {*edenot loc s′*}
| (*s*, *h*), (*s′*, *h′*) ⊨ *Exists x A* ⟷ (∃ *v v′*. (*s*(*x* := *v*), *h*), (*s′*(*x* := *v′*), *h′*) ⊨ *A*)

| (*s*, *h*), (*s′*, *h′*) ⊨ *EmptyFullGuards* ⟷ (*get-gs h* = *Some* (*pwrite*, {#}) ∧ (∀ *k*. *get-gu h k* = *Some* []) ∧ *get-gs h′* = *Some* (*pwrite*, {#}) ∧ (∀ *k*. *get-gu h′ k* = *Some* []))

| (*s*, *h*), (*s′*, *h′*) ⊨ *PreSharedGuards spre* ⟷
  (∃ *sargs sargs′*. *get-gs h* = *Some* (*pwrite*, *sargs*) ∧ *get-gs h′* = *Some* (*pwrite*, *sargs′*) ∧ *PRE-shared-simpler* (*Rep-precondition spre*) *sargs sargs′*
  ∧ *get-fh h* = *Map.empty* ∧ *get-fh h′* = *Map.empty*)
| (*s*, *h*), (*s′*, *h′*) ⊨ *PreUniqueGuards upre* ⟷
  (∃ *uargs uargs′*. (∀ *k*. *get-gu h k* = *Some* (*uargs k*)) ∧ (∀ *k*. *get-gu h′ k* = *Some* (*uargs′ k*)) ∧ (∀ *k*. *PRE-unique* (*Rep-indexed-precondition upre k*) (*uargs k*) (*uargs′ k*)) ∧ *get-fh h* = *Map.empty* ∧ *get-fh h′* = *Map.empty*)

| (*s*, *h*), (*s′*, *h′*) ⊨ *View f J e* ⟷ ((*s*, *h*), (*s′*, *h′*) ⊨ *J* ∧ *f* (*normalize* (*get-fh h*)) = *e s* ∧ *f* (*normalize* (*get-fh h′*)) = *e s′*)
| (*s*, *h*), (*s′*, *h′*) ⊨ *SharedGuard π ms* ⟷ ((∀ *k*. *get-gu h k* = *None* ∧ *get-gu h′ k* = *None*) ∧ *get-gs h* = *Some* (*π*, *ms s*) ∧ *get-gs h′* = *Some* (*π*, *ms s′*)
    ∧ *get-fh h* = *Map.empty* ∧ *get-fh h′* = *Map.empty*)

| (*s*, *h*), (*s′*, *h′*) ⊨ *UniqueGuard k lexp* ⟷ (*get-gs h* = *None* ∧ *get-gu h k* = *Some* (*lexp s*) ∧ *get-gu h′ k* = *Some* (*lexp s′*) ∧ *get-gs h′* = *None*
    ∧ *get-fh h* = *Map.empty* ∧ *get-fh h′* = *Map.empty* ∧ (∀ *k′*. *k′* ≠ *k* ⟶ *get-gu h k′* = *None* ∧ *get-gu h′ k′* = *None*))

| (*s*, *h*), (*s′*, *h′*) ⊨ *LowExp e* ⟷ *edenot e s* = *edenot e s′*

| (*s*, *h*), (*s′*, *h′*) ⊨ *Imp b A* ⟷ *bdenot b s* = *bdenot b s′* ∧ (*bdenot b s* ⟶ (*s*, *h*), (*s′*, *h′*) ⊨ *A*)

| (*s*, *h*), (*s′*, *h′*) ⊨ *NoGuard* ⟷ (*get-gs h* = *None* ∧ (∀ *k*. *get-gu h k* = *None*) ∧ *get-gs h′* = *None* ∧ (∀ *k*. *get-gu h′ k* = *None*))

lemma *sat-PreUniqueE*:
  assumes (*s*, *h*), (*s′*, *h′*) ⊨ *PreUniqueGuards upre*

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

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

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

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

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

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

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

definition *subS* :: *var* ⇒ *exp* ⇒ (*store* ⇒ *'v*) ⇒ (*store* ⇒ *'v*) **where**
subS x E e = (λ*s*. *e* (*s*(*x* := *edenot E s*)))

lemma *subS-assign*:

subS x E e s $\longleftrightarrow$ *e* (*s*(*x* := *edenot E s*))
⟨*proof*⟩

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

fun *subA* :: *var* ⇒ *exp* ⇒ ('*i*, '*a*, *nat*) *assertion* ⇒ ('*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* ∩ *fvE E* = {}
shows (*s*, *h*), (*s*′, *h*′) ⊨ *subA x E A* $\longleftrightarrow$ (*s*(*x* := *edenot E s*), *h*), (*s*′(*x* := *edenot E s*′), *h*′) ⊨ *A*
⟨*proof*⟩

lemma *PRE-uniqueI*:
assumes *length uargs* = *length uargs*′
and ⋀*i*. *i* ≥ *0* ∧ *i* < *length uargs*′ ⟹ *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* ∧ *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* ∧ *PRE-shared-simpler* (*Rep-precondition pre*) *a a*
⟨*proof*⟩

lemma *always-sat-refl-aux*:
assumes (*s*, *h*), (*s′*, *h′*) ⊨ *A*
shows (*s*, *h*), (*s*, *h*) ⊨ *A*
⟨*proof*⟩

lemma *always-sat-refl*:
assumes *σ*, *σ′* ⊨ *A*
shows *σ*, *σ* ⊨ *A*
⟨*proof*⟩

lemma *agrees-same-aux*:
assumes *agrees* (*fvA A*) *s s″*
and (*s*, *h*), (*s′*, *h′*) ⊨ *A*
shows (*s″*, *h*), (*s′*, *h′*) ⊨ *A*
⟨*proof*⟩

lemma *agrees-same*:
assumes *agrees* (*fvA A*) *s s″*
shows (*s*, *h*), (*s′*, *h′*) ⊨ *A* ⟷ (*s″*, *h*), (*s′*, *h′*) ⊨ *A*
⟨*proof*⟩

lemma *sat-comm-aux*:
(*s*, *h*), (*s′*, *h′*) ⊨ *A* ⟹ (*s′*, *h′*), (*s*, *h*) ⊨ *A*
⟨*proof*⟩

lemma *sat-comm*:
σ, *σ′* ⊨ *A* ⟷ *σ′*, *σ* ⊨ *A*
⟨*proof*⟩

definition *precise* **where**
precise J ⟷ (∀ *s1 H1 h1 h1′ s2 H2 h2 h2′*. *H1* ≽ *h1* ∧ *H1* ≽ *h1′* ∧ *H2* ≽ *h2* ∧ *H2* ≽ *h2′*
∧ (*s1*, *h1*), (*s2*, *h2*) ⊨ *J* ∧ (*s1*, *h1′*), (*s2*, *h2′*) ⊨ *J* ⟶ *h1′* = *h1* ∧ *h2′* = *h2*)

lemma *preciseI*:
assumes ⋀*s1 H1 h1 h1′ s2 H2 h2 h2′*. *H1* ≽ *h1* ∧ *H1* ≽ *h1′* ∧ *H2* ≽ *h2* ∧ *H2* ≽ *h2′* ⟹
(*s1*, *h1*), (*s2*, *h2*) ⊨ *J* ⟹ (*s1*, *h1′*), (*s2*, *h2′*) ⊨ *J* ⟹ *h1′* = *h1* ∧ *h2′* = *h2*
shows *precise J*
⟨*proof*⟩

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* $\Longrightarrow$ (*s1*, *h1*), (*s2*, *h2*) $\models$ *J*
shows *unary J*
⟨*proof*⟩

lemma *unary-smallerI*:
assumes $\bigwedge$*σ1 σ2*. *σ1*, *σ1* $\models$ *J* $\wedge$ *σ2*, *σ2* $\models$ *J* $\Longrightarrow$ *σ1*, *σ2* $\models$ *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 $\longleftrightarrow$ ($\forall$*σ σ'*. *σ*, *σ'* $\models$ *A* $\longrightarrow$ *σ*, *σ'* $\models$ *B*)

lemma *entailsI*:
assumes $\bigwedge$*x y*. *x*, *y* $\models$ *A* $\Longrightarrow$ *x*, *y* $\models$ *B*
shows *entails A B*
⟨*proof*⟩

lemma *sat-points-to*:
assumes (*s*, *h* :: ('*i*, '*a*) *heap*), (*s*, *h*) $\models$ *PointsTo a π e*
shows *get-fh h* = [*edenot a s* $\mapsto$ (*π*, *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* ⇒ *bool* **where**
syntactic-unary (*Bool b*) ⟷ *True*
| *syntactic-unary* (*And A B*) ⟷ *syntactic-unary A* ∧ *syntactic-unary B*
| *syntactic-unary* (*Star A B*) ⟷ *syntactic-unary A* ∧ *syntactic-unary B*
| *syntactic-unary* (*Low e*) ⟷ *False*
| *syntactic-unary Emp* ⟷ *True*
| *syntactic-unary* (*PointsTo v va vb*) ⟷ *True*
| *syntactic-unary* (*Exists x A*) ⟷ *syntactic-unary A*
| *syntactic-unary* (*SharedGuard - e*) ⟷ *True*
| *syntactic-unary* (*UniqueGuard - e*) ⟷ *True*
| *syntactic-unary* (*View view A e*) ⟷ *syntactic-unary A*
| *syntactic-unary* (*PreSharedGuards -*) ⟷ *False*
| *syntactic-unary* (*PreUniqueGuards -*) ⟷ *False*
| *syntactic-unary EmptyFullGuards* ⟷ *True*
| *syntactic-unary* (*LowExp x*) ⟷ *False*
| *syntactic-unary* (*Imp b A*) ⟷ *False*

lemma *syntactic-unary-implies-unary*:
assumes *syntactic-unary A*
shows *unary A*
⟨*proof*⟩

record ('*i*, '*a*, '*v*) *single-context* =
view :: (*loc* ⇀ *val*) ⇒ '*v*
abstract-view :: '*v* ⇒ '*v*
saction :: '*v* ⇒ '*a* ⇒ '*v*
uaction :: '*i* ⇒ '*v* ⇒ '*a* ⇒ '*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 ⟷ (∀ *s1 h1 s2 h2*. (*s1*, *h1*), (*s2*, *h2*) ⊨ *A* ⟶ *no-guard h1* ∧ *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* ⇒ *bool* **where**
view-function-of-inv Γ ⟷ (∀ (*h* :: ('*i*, '*a*) *heap*) (*h'* :: ('*i*, '*a*) *heap*) *s*. (*s*, *h*), (*s*, *h*) ⊨ *invariant* Γ ∧ (*h'* ⪰ *h*)
⟶ *view* Γ (*normalize* (*get-fh h*)) = *view* Γ (*normalize* (*get-fh h'*)))

definition *wf-indexed-precondition* :: $('i \Rightarrow 'a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
wf-indexed-precondition pre $\longleftrightarrow$ $(\forall\, a\ b\ k.\ pre\ k\ a\ b \longrightarrow (pre\ k\ b\ a \wedge pre\ k\ a\ a))$

definition *wf-precondition* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
wf-precondition pre $\longleftrightarrow$ $(\forall\, a\ b.\ pre\ a\ b \longrightarrow (pre\ b\ a \wedge pre\ a\ a))$

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* (λ*s*. *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* :: $('a\ set) \Rightarrow ('a\ set) \Rightarrow bool$
where *disjoint h1 h2* = (*h1* ∩ *h2* = {})

definition *unambiguous* **where**
unambiguous A x $\longleftrightarrow$ $(\forall\, s1\ h1\ s2\ h2\ v1\ v2\ v1'\ v2'.\ (s1(x := v1), h1), (s2(x := v2), h2) \models A$
$\wedge\ (s1(x := v1'), h1), (s2(x := v2'), h2) \models A \longrightarrow v1 = v1' \wedge v2 = v2')$

definition *all-axioms* :: $('v \Rightarrow 'w) \Rightarrow ('v \Rightarrow 'a \Rightarrow 'v) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('i \Rightarrow 'v \Rightarrow 'b \Rightarrow 'v) \Rightarrow ('i \Rightarrow 'b \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$ **where**
all-axioms α *sact spre uact upre* $\longleftrightarrow$

— Every action's relational precondition is sufficient to preserve the low-ness of the abstract view of the resource value:
(∀ v v' *sarg sarg'*. α $v = \alpha$ $v' \wedge$ *spre sarg sarg'* $\longrightarrow \alpha$ (*sact v sarg*) $= \alpha$ (*sact v' sarg'*)) $\wedge$
(∀ v v' *uarg uarg' i*. α $v = \alpha$ $v' \wedge$ *upre i uarg uarg'* $\longrightarrow \alpha$ (*uact i v uarg*) $= \alpha$ (*uact i v' uarg'*)) $\wedge$

(∀ *sarg sarg'*. *spre sarg sarg'* $\longrightarrow$ *spre sarg' sarg'*) $\wedge$
(∀ *uarg uarg' i*. *upre i uarg uarg'* $\longrightarrow$ *upre i uarg' uarg'*) $\wedge$

— All relevant pairs of actions commute w.r.t. the abstract view:
(∀ v v' *sarg sarg'*. α $v = \alpha$ $v' \wedge$ *spre sarg sarg* $\wedge$ *spre sarg' sarg'* $\longrightarrow \alpha$ (*sact* (*sact v sarg*) *sarg'*) $= \alpha$ (*sact* (*sact v' sarg'*) *sarg*)) $\wedge$
(∀ v v' *sarg uarg i*. α $v = \alpha$ $v' \wedge$ *spre sarg sarg* $\wedge$ *upre i uarg uarg* $\longrightarrow \alpha$ (*sact* (*uact i v uarg*) *sarg*) $= \alpha$ (*uact i* (*sact v' sarg*) *uarg*)) $\wedge$
(∀ 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*))

3.2 Rules of the Logic

inductive *CommCSL* :: ('*i*, '*a*, *nat*) *cont* ⇒ ('*i*, '*a*, *nat*) *assertion* ⇒ *cmd* ⇒ ('*i*, '*a*, *nat*) *assertion* ⇒ *bool*
(- ⊢ {-} - {-} [*51*,*0*,*0*] *81*) **where**
RuleSkip: Δ ⊢ {*P*} *Cskip* {*P*}
| *RuleAssign*: ⟦ ⋀Γ. Δ = *Some* Γ ⟹ *x* ∉ *fvA* (*invariant* Γ) ; *collect-existentials P* ∩ *fvE E* = {} ⟧ ⟹ Δ ⊢ {*subA x E P*} *Cassign x E* {*P*}
| *RuleNew*: ⟦ *x* ∉ *fvE E*; ⋀Γ. Δ = *Some* Γ ⟹ *x* ∉ *fvA* (*invariant* Γ) ∧ *view-function-of-inv* Γ ⟧ ⟹ Δ ⊢ {*Emp*} *Calloc x E* {*PointsTo* (*Evar x*) *pwrite E*}
| *RuleWrite*: ⟦ ⋀Γ. Δ = *Some* Γ ⟹ *view-function-of-inv* Γ ; *v* ∉ *fvE loc* ⟧
⟹ Δ ⊢ {*Exists v* (*PointsTo loc pwrite* (*Evar v*))} *Cwrite loc E* {*PointsTo loc pwrite E*}
| ⟦ ⋀Γ. Δ = *Some* Γ ⟹ *x* ∉ *fvA* (*invariant* Γ) ∧ *view-function-of-inv* Γ ; *x* ∉ *fvE E* ∪ *fvE e* ⟧ ⟹
Δ ⊢ {*PointsTo E* π *e*} *Cread x E* {*And* (*PointsTo E* π *e*) (*Bool* (*Beq* (*Evar x*) *e*))}
| *RuleShare*: ⟦ Γ = ⦇ *view* = *f*, *abstract-view* = α, *saction* = *sact*, *uaction* = *uact*, *invariant* = *J* ⦈ ; *all-axioms* α *sact spre uact upre* ;
Some Γ ⊢ {*Star P EmptyFullGuards*} *C* {*Star Q* (*And* (*PreSharedGuards* (*Abs-precondition spre*)) (*PreUniqueGuards* (*Abs-indexed-precondition upre*)))};
view-function-of-inv Γ ; *unary J* ; *precise J* ; *wf-indexed-precondition upre* ; *wf-precondition spre* ; *x* ∉ *fvA J* ;
no-guard-assertion (*Star P* (*LowView* (α ∘ *f*) *J x*)) ⟧ ⟹ *None* ⊢ {*Star P* (*LowView* (α ∘ *f*) *J x*)} *C* {*Star Q* (*LowView* (α ∘ *f*) *J x*)}
| *RuleAtomicUnique*: ⟦ Γ = ⦇ *view* = *f*, *abstract-view* = α, *saction* = *sact*, *uaction* = *uact*, *invariant* = *J* ⦈ ;
no-guard-assertion P ; *no-guard-assertion Q* ;
None ⊢ {*Star P* (*View f J* (λ*s*. *s x*))} *C* {*Star Q* (*View f J* (λ*s*. *uact index* (*s x*) (*map-to-arg* (*s uarg*))))} ;

precise J ; *unary J* ; *view-function-of-inv* Γ ; $x \notin$ *fvC C* ∪ *fvA P* ∪ *fvA Q* ∪ *fvA J* ; *uarg* ∉ *fvC C* ;
$l \notin$ *fvC C* ; $x \notin$ *fvS* (λ*s*. *map-to-list* (*s l*)) ; $x \notin$ *fvS* (λ*s*. *map-to-arg* (*s uarg*) # *map-to-list* (*s l*)) ⟧
⟹ *Some* Γ ⊢ {*Star P* (*UniqueGuard index* (λ*s*. *map-to-list* (*s l*)))} *Catomic C* {*Star Q* (*UniqueGuard index* (λ*s*. *map-to-arg* (*s uarg*) # *map-to-list* (*s l*)))}
| *RuleAtomicShared*: ⟦ Γ = (| *view* = *f*, *abstract-view* = α, *saction* = *sact*, *uaction* = *uact*, *invariant* = *J* |) ; *no-guard-assertion P* ; *no-guard-assertion Q* ;
None ⊢ {*Star P* (*View f J* (λ*s*. *s x*))} *C* {*Star Q* (*View f J* (λ*s*. *sact* (*s x*) (*map-to-arg* (*s sarg*))))} ;
precise J ; *unary J* ; *view-function-of-inv* Γ ; $x \notin$ *fvC C* ∪ *fvA P* ∪ *fvA Q* ∪ *fvA J* ; *sarg* ∉ *fvC C* ;
ms ∉ *fvC C* ; $x \notin$ *fvS* (λ*s*. *map-to-multiset* (*s ms*)) ; $x \notin$ *fvS* (λ*s*. {# *map-to-arg* (*s sarg*) #} + *map-to-multiset* (*s ms*)) ⟧
⟹ *Some* Γ ⊢ {*Star P* (*SharedGuard* π (λ*s*. *map-to-multiset* (*s ms*)))} *Catomic C* {*Star Q* (*SharedGuard* π (λ*s*. {# *map-to-arg* (*s sarg*) #} + *map-to-multiset* (*s ms*)))}
| *RulePar*: ⟦ Δ ⊢ {*P1*} *C1* {*Q1*} ; Δ ⊢ {*P2*} *C2* {*Q2*} ; *disjoint* (*fvA P1* ∪ *fvC C1* ∪ *fvA Q1*) (*wrC C2*) ;
disjoint (*fvA P2* ∪ *fvC C2* ∪ *fvA Q2*) (*wrC C1*) ; ⋀Γ. Δ = *Some* Γ ⟹ *disjoint* (*fvA* (*invariant* Γ)) (*wrC C2*) ;
⋀Γ. Δ = *Some* Γ ⟹ *disjoint* (*fvA* (*invariant* Γ)) (*wrC C1*) ; *precise P1* ∨ *precise P2* ⟧
⟹ Δ ⊢ {*Star P1 P2*} *Cpar C1 C2* {*Star Q1 Q2*}
| *RuleIf1*: ⟦ Δ ⊢ {*And P* (*Bool b*)} *C1* {*Q*} ; Δ ⊢ {*And P* (*Bool* (*Bnot b*))} *C2* {*Q*} ⟧
⟹ Δ ⊢ {*And P* (*Low b*)} *Cif b C1 C2* {*Q*}
| *RuleIf2*: ⟦ Δ ⊢ {*And P* (*Bool b*)} *C1* {*Q*} ; Δ ⊢ {*And P* (*Bool* (*Bnot b*))} *C2* {*Q*} ; *unary Q* ⟧
⟹ Δ ⊢ {*P*} *Cif b C1 C2* {*Q*}
| *RuleSeq*: ⟦ Δ ⊢ {*P*} *C1* {*R*} ; Δ ⊢ {*R*} *C2* {*Q*} ⟧ ⟹ Δ ⊢ {*P*} *Cseq C1 C2* {*Q*}
| *RuleFrame*: ⟦ Δ ⊢ {*P*} *C* {*Q*} ; *disjoint* (*fvA R*) (*wrC C*) ; *precise P* ∨ *precise R* ⟧
⟹ Δ ⊢ {*Star P R*} *C* {*Star Q R*}
| *RuleCons*: ⟦ Δ ⊢ {*P'*} *C* {*Q'*} ; *entails P P'* ; *entails Q' Q* ⟧ ⟹ Δ ⊢ {*P*} *C* {*Q*}
| *RuleExists*: ⟦ Δ ⊢ {*P*} *C* {*Q*} ; $x \notin$ *fvC C* ; ⋀Γ. Δ = *Some* Γ ⟹ $x \notin$ *fvA* (*invariant* Γ) ; *unambiguous P x* ⟧
⟹ Δ ⊢ {*Exists x P*} *C* {*Exists x Q*}
| *RuleWhile1*: Δ ⊢ {*And I* (*Bool b*)} *C* {*And I* (*Low b*)} ⟹ Δ ⊢ {*And I* (*Low b*)} *Cwhile b C* {*And I* (*Bool* (*Bnot b*))}
| *RuleWhile2*: ⟦ *unary I* ; Δ ⊢ {*And I* (*Bool b*)} *C* {*I*} ⟧ ⟹ Δ ⊢ {*I*} *Cwhile b C* {*And I* (*Bool* (*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'* $\Longrightarrow \alpha$ (*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'* $\Longrightarrow \alpha$ (*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'* $\Longrightarrow$ *spre sarg' sarg'*
⟨*proof*⟩

lemma *upre-refl*:
assumes *all-axioms* α *sact spre uact upre*
shows *upre i uarg uarg'* $\Longrightarrow$ *upre i uarg' uarg'*
⟨*proof*⟩

lemma *ss-com*:
assumes *all-axioms* α *sact spre uact upre*
shows $\alpha\ v = \alpha\ v' \Longrightarrow$ *spre sarg sarg* $\wedge$ *spre sarg' sarg'* $\Longrightarrow \alpha$ (*sact* (*sact v sarg*) *sarg'*) $= \alpha$ (*sact* (*sact v' sarg'*) *sarg*)
⟨*proof*⟩

lemma *su-com*:
assumes *all-axioms* α *sact spre uact upre*
shows $\alpha\ v = \alpha\ v' \Longrightarrow$ *spre sarg sarg* $\wedge$ *upre i uarg uarg* $\Longrightarrow \alpha$ (*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 α (*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 ⋀*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* ≥ *0* ∧ *i* < *n* ⟹ *remove-at-index n l* ! *i* = *l* ! *i*
and *i* ≥ *n* ∧ *i* < *length l* − *1* ⟹ *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* ≤ *length l*

shows *length* (*insert-at n x l*) = *length l* + *1*
and $i \geq 0 \wedge i < n \Longrightarrow$ *insert-at n x l* ! *i* = *l* ! *i*
and $n \geq 0 \Longrightarrow$ *insert-at n x l* ! *n* = *x*
and $i > n \wedge i <$ *length l* + *1* $\Longrightarrow$ *insert-at n x l* ! *i* = *l* ! (*i* − *1*)
⟨*proof*⟩

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

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

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

lemma *exists-index-in-sequence-shared*:
assumes *a* ∈# *sargs*
and *possible-sequence sargs uargs s*
shows ∃ *i*. *i* ≥ *0* ∧ *i* < *length s* ∧ *s* ! *i* = *Shared a* ∧ *possible-sequence* (*sargs* − {# *a* #}) *uargs* (*remove-at-index i s*)
⟨*proof*⟩

lemma *head-possible-exists-first-unique*:
assumes *a* = *hd* (*uargs j*)
and *uargs j* ≠ []
and *possible-sequence sargs uargs s*
shows ∃ *i*. *i* ≥ *0* ∧ *i* < *length s* ∧ *s* ! *i* = *Unique j a* ∧ (∀ *k*. *k* ≥ *0* ∧ *k* < *i* ⟶ ¬ *is-Unique-i j* (*s* ! *k*))
⟨*proof*⟩

lemma *remove-at-index-filter*:
assumes $i \geq 0 \wedge i <$ *length s* ∧ *P* (*s* ! *i*)
and $\bigwedge j.\ j \geq 0 \wedge j < i \Longrightarrow$ ¬ *P* (*s* ! *j*)
shows *tl* (*map get-u* (*filter P s*)) = *map get-u* (*filter P* (*remove-at-index i s*))
⟨*proof*⟩

definition *tail-kth* **where**
tail-kth uargs k = *uargs*(*k* := *tl* (*uargs k*))

lemma *exists-index-in-sequence-unique*:
assumes *a* = *hd* (*uargs k*)

and *uargs k* $\neq$ []
and *possible-sequence sargs uargs s*
shows $\exists$ *i. i* $\geq$ *0* $\wedge$ *i* $<$ *length s* $\wedge$ *s* ! *i* = *Unique k a* $\wedge$ *possible-sequence sargs* (*tail-kth uargs k*) (*remove-at-index i s*)
$\wedge$ ($\forall$ *j. j* $\geq$ *0* $\wedge$ *j* $<$ *i* $\longrightarrow$ $\neg$ *is-Unique-i k* (*s* ! *j*))
⟨*proof*⟩

lemma *possible-sequence-where-is-unique*:
assumes *possible-sequence sargs uargs* (*Unique k a* # *s*)
shows *a* = *hd* (*uargs k*)
⟨*proof*⟩

lemma *possible-sequence-where-is-shared*:
assumes *possible-sequence sargs uargs* (*Shared a* # *s*)
shows *a* $\in\#$ *sargs*
⟨*proof*⟩

lemma *PRE-unique-tlI*:
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* => *'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* $\Longrightarrow$ *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 ∃ *sarg′*. *sarg′* ∈# *sargs′* ∧ *spre sarg sarg′* ∧ *PRE-shared spre* (*sargs* − {# *sarg* #}) (*sargs′* − {# *sarg′* #})
⟨*proof*⟩

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

and *PRE-shared spre sargs sargs′*
and ⋀*k*. *PRE-unique* (*upre k*) (*uargs k*) (*uargs′ k*)

shows ∃ *s″*. *possible-sequence sargs′ uargs′ s″* ∧ *PRE-sequence spre upre s s″*
⟨*proof*⟩

lemma *insert-remove-same-list*:
assumes *k* ≥ *0* ∧ *k* < *length s*
and *s* ! *k* = *x*
shows *s* = *insert-at k x* (*remove-at-index k s*)
⟨*proof*⟩

lemma *swap-works*:
assumes *length s* = *length s′*
and *k* < *length s* − *1*
and ⋀*i*. *i* ≥ *0* ∧ *i* < *length s* ∧ *i* ≠ *k* ∧ *i* ≠ *k* + *1* ⟹ *s* ! *i* = *s′* ! *i*
and *s* ! *k* = *s′* ! (*k* + *1*)
and *s′* ! *k* = *s* ! (*k* + *1*)
and *PRE-sequence spre upre s s*
and *α v0* = *α v0′*

and $\neg\ (\exists\, k'.\ \textit{is-Unique-i}\ k'\ (s\ !\ k) \wedge \textit{is-Unique-i}\ k'\ (s\ !\ (k + 1)))$
and *all-axioms* α *sact spre uact upre*
shows α (*compute sact uact v0 s*) = α (*compute sact uact v0′ s′*) (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *mset-remove*:
assumes $k \geq 0 \wedge k <$ *length s*
shows *mset s* = *mset* (*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 α *v0* = α *v0′*
and *PRE-sequence spre upre s s*
and *PRE-sequence spre upre s′ s′*
and *all-axioms* α *sact spre uact upre*
shows α (*compute sact uact v0 s*) = α (*compute sact uact v0′ s′*)
⟨*proof*⟩

lemma *PRE-sequence-same-abstract*:
assumes *PRE-sequence spre upre s s'*
and α *v0* = α *v0'*
and *all-axioms* α *sact spre uact upre*
shows α (*compute sact uact v0 s*) = α (*compute sact uact v0' s'*)
⟨*proof*⟩

lemma *simple-possible-PRE-seq*:
assumes *possible-sequence sargs uargs s*
and *possible-sequence sargs' uargs' s'*
and *PRE-shared spre sargs sargs'*
and $\bigwedge$*k. PRE-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. PRE-unique* (*upre k*) (*uargs k*) (*uargs' k*)

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

shows α (*compute sact uact v0 s*) = α (*compute sact uact v0' s'*)
⟨*proof*⟩

inductive *reachable-value* :: ('*v* ⇒ '*a* ⇒ '*v*) ⇒ ('*i* ⇒ '*v* ⇒ '*b* ⇒ '*v*) ⇒ '*v* ⇒ '*a multiset* ⇒ ('*i* ⇒ '*b list*) ⇒ '*v* ⇒ *bool* **where**
Self: *reachable-value sact uact v0* {#} (λ*k*. []) *v0*
| *SharedStep*: *reachable-value sact uact v0 sargs uargs v1* ⟹ *reachable-value sact uact v0* (*sargs* + {# *sarg* #}) *uargs* (*sact v1 sarg*)
| *UniqueStep*: *reachable-value sact uact v0 sargs uargs v1* ⟹ *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 ∃ *s. possible-sequence sargs uargs s* ∧ *v1* = *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 $\bigwedge k$. *PRE-unique* (*upre k*) (*uargs k*) (*uargs' k*)
and α *v0* = α *v0'*
and *all-axioms* α *sact spre uact upre*
shows α *v* = α *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* ⇒ '*v* ⇒ ('*i*, '*a*) *heap* ⇒ *bool* **where**
reachable scont v0 h ⟷ (∀ *sargs uargs*. *get-gs h* = *Some* (*pwrite*, *sargs*) ∧ (∀ *k*. *get-gu h k* = *Some* (*uargs k*))
⟶ *reachable-value* (*saction scont*) (*uaction scont*) *v0 sargs uargs* (*view scont* (*normalize* (*get-fh h*))))

lemma *reachableI*:
assumes $\bigwedge$*sargs uargs*. *get-gs h* = *Some* (*pwrite*, *sargs*) ∧ (∀ *k*. *get-gu h k* = *Some* (*uargs k*))
⟹ *reachable-value* (*saction scont*) (*uaction scont*) *v0 sargs uargs* (*view scont* (*normalize* (*get-fh h*)))
shows *reachable scont v0 h*
⟨*proof*⟩

lemma *reachableE*:
assumes *reachable scont v0 h*
and *get-gs h* = *Some* (*pwrite*, *sargs*)
and $\bigwedge k$. *get-gu h k* = *Some* (*uargs k*)
shows *reachable-value* (*saction scont*) (*uaction scont*) *v0 sargs uargs* (*view scont* (*normalize* (*get-fh h*)))
⟨*proof*⟩

definition *all-guards* :: ('*i*, '*a*) *heap* ⇒ *bool* **where**
all-guards h ⟷ (∃ *v*. *get-gs h* = *Some* (*pwrite*, *v*)) ∧ (∀ *k*. *get-gu h k* ≠ *None*)

lemma *no-guardI*:
assumes *get-gs h* = *None*
and $\bigwedge k$. *get-gu h k* = *None*
shows *no-guard h*

⟨*proof*⟩

definition *semi-consistent* :: ('*i*, '*a*, '*v*) *single-context* ⇒ '*v* ⇒ ('*i*, '*a*) *heap* ⇒ *bool* **where**
semi-consistent Γ *v0 h* ⟷ *all-guards h* ∧ *reachable* Γ *v0 h*

lemma *semi-consistentE*:
assumes *semi-consistent* Γ *v0 h*
shows ∃ *sargs uargs*. *get-gs h* = *Some* (*pwrite*, *sargs*) ∧ (∀ *k*. *get-gu h k* = *Some* (*uargs k*))
∧ *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* ⊕ *Some b*
and *no-guard h*
shows *no-guard a*
⟨*proof*⟩

lemma *all-guardsI*:
assumes ⋀*k*. *get-gu h k* ≠ *None*
and ∃ *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* ⊕ *Some b*
shows *all-guards h*
⟨*proof*⟩

definition *empty-unique* **where**
empty-unique - = *None*

definition *remove-guards* :: ('*i*, '*a*) *heap* ⇒ ('*i*, '*a*) *heap* **where**
remove-guards h = (*get-fh h*, *None*, *empty-unique*)

lemma *remove-guards-smaller*:
h ≽ *remove-guards h*
⟨*proof*⟩

lemma *no-guard-remove*:

assumes *Some a = Some b ⊕ Some c*
and *no-guard c*
shows *get-gs a = get-gs b*
and *get-gu a = get-gu b*
⟨*proof*⟩

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

lemma *sum-of-no-guards*:
assumes *no-guard a*
and *no-guard b*
and *Some x = Some a ⊕ Some b*
shows *no-guard x*
⟨*proof*⟩

lemma *no-guard-remove-guards*:
no-guard (*remove-guards h*)
⟨*proof*⟩

lemma *get-fh-remove-guards*:
get-fh (*remove-guards h*) = *get-fh h*
⟨*proof*⟩

definition *pair-sat* :: (*store* × ('*i*, '*a*) *heap*) *set* ⇒ (*store* × ('*i*, '*a*) *heap*) *set* ⇒ ('*i*, '*a*, *nat*) *assertion* ⇒ *bool* **where**
pair-sat S S' Q ⟷ (∀ σ σ'. σ ∈ *S* ∧ σ' ∈ *S'* ⟶ σ, σ' ⊨ *Q*)

lemma *pair-satI*:
assumes ⋀*s h s' h'*. (*s*, *h*) ∈ *S* ∧ (*s'*, *h'*) ∈ *S'* ⟹ (*s*, *h*), (*s'*, *h'*) ⊨ *Q*
shows *pair-sat S S' Q*
⟨*proof*⟩

lemma *pair-sat-smallerI*:
assumes ⋀σ σ'. σ ∈ *S* ∧ σ' ∈ *S'* ⟹ σ, σ' ⊨ *Q*
shows *pair-sat S S' Q*
⟨*proof*⟩

lemma *pair-satE*:
assumes *pair-sat S S' Q*
and (*s*, *h*) ∈ *S* ∧ (*s'*, *h'*) ∈ *S'*
shows (*s*, *h*), (*s'*, *h'*) ⊨ *Q*
⟨*proof*⟩

definition *add-states* :: (*store* × ('*i*, '*a*) *heap*) *set* ⇒ (*store* × ('*i*, '*a*) *heap*) *set* ⇒ (*store* × ('*i*, '*a*) *heap*) *set* **where**

add-states S1 S2 = {(*s*, *H*) |*s H h1 h2*. *Some H* = *Some h1* ⊕ *Some h2* ∧ (*s*, *h1*) ∈ *S1* ∧ (*s*, *h2*) ∈ *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* ⊆ *S1′*
shows *add-states S1 S2* ⊆ *add-states S1′ S2*
⟨*proof*⟩

lemma *add-states-comm*:
add-states S1 S2 = *add-states S2 S1*
⟨*proof*⟩

lemma *magic-lemma*:
assumes *Some x1* = *Some a1* ⊕ *Some j1*
and *Some x2* = *Some a2* ⊕ *Some j2*
and (*s1*, *x1*), (*s2*, *x2*) ⊨ *Star A J*
and (*s1*, *j1*), (*s2*, *j2*) ⊨ *J*
and *precise J*
shows (*s1*, *a1*), (*s2*, *a2*) ⊨ *A*
⟨*proof*⟩

lemma *full-no-guard-same-normalize*:
assumes *full-ownership* (*get-fh h*) ∧ *no-guard h*
and *full-ownership* (*get-fh h′*) ∧ *no-guard h′*
and *normalize* (*get-fh h*) = *normalize* (*get-fh h′*)
shows *h* = *h′*
⟨*proof*⟩

lemma *get-fh-same-then-remove-guards-same*:
assumes *get-fh a* = *get-fh b*
shows *remove-guards a* = *remove-guards b*
⟨*proof*⟩

lemma *remove-guards-sum*:
assumes *Some x* = *Some a* ⊕ *Some b*
shows *Some* (*remove-guards x*) = *Some* (*remove-guards a*) ⊕ *Some* (*remove-guards b*)
⟨*proof*⟩

lemma *no-guard-smaller*:
assumes *a* ≽ *b*

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

definition *add-empty-guards* :: (*'i*, *'a*) *heap* ⇒ (*'i*, *'a*) *heap* **where**
add-empty-guards h = (*get-fh h*, *Some* (*pwrite*, {#}), (λ-. *Some* []))

lemma *no-guard-map-empty-compatible*:
assumes *no-guard a*
and *get-fh b* = *Map.empty*
shows *a* ## *b*
⟨*proof*⟩

lemma *no-guard-add-empty-is-add*:
assumes *no-guard h*
shows *Some* (*add-empty-guards h*) = *Some h* ⊕ *Some* (*Map.empty*, *Some* (*pwrite*, {#}), (λ-. *Some* []))
⟨*proof*⟩

lemma *no-guard-and-sat-p-empty-guards*:
assumes (*s*, *h*), (*s'*, *h'*) ⊨ *A*
and *no-guard h* ∧ *no-guard h'*
shows (*s*, *add-empty-guards h*), (*s'*, *add-empty-guards h'*) ⊨ *Star A EmptyFull-Guards*
⟨*proof*⟩

lemma *no-guard-add-empty-guards-sum*:
assumes *no-guard x*
and *Some x* = *Some a* ⊕ *Some b*
shows *Some* (*add-empty-guards x*) = *Some* (*add-empty-guards a*) ⊕ *Some b*
⟨*proof*⟩

lemma *semi-consistent-empty-no-guard-initial-value*:
assumes *no-guard h*
shows *semi-consistent* Γ (*view* Γ (*FractionalHeap.normalize* (*get-fh h*))) (*add-empty-guards h*)
⟨*proof*⟩

lemma *no-guards-remove-same*:
assumes *no-guard h*
shows *h* = *remove-guards* (*add-empty-guards h*)
⟨*proof*⟩

lemma *no-guards-remove*:
no-guard h ⟷ *h* = *remove-guards h*
⟨*proof*⟩

definition *add-sguard-to-no-guard* :: (*'i*, *'a*) *heap* ⇒ *prat* ⇒ *'a multiset* ⇒ (*'i*, *'a*) *heap* **where**
add-sguard-to-no-guard h π *ms* = (*get-fh h*, *Some* (π, *ms*), (λ-. *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* $\Longrightarrow$ *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 *Some x* = *Some a* ⊕ *Some b*
and *no-guard a*
shows *all-guards b*
⟨*proof*⟩

lemma *all-guards-exists-uargs*:
assumes *all-guards x*
shows ∃ *uargs*. ∀ *k*. *get-gu x k* = *Some* (*uargs k*)
⟨*proof*⟩

lemma *all-guards-sum-known-one*:
assumes *Some x* = *Some a* ⊕ *Some b*
and *all-guards x*
and ⋀*k*. *get-gu a k* = *None*
and *get-gs a* = *Some* (π, *ms*)
shows ∃ π′ *msf uargs*. (∀ *k*. *get-gu b k* = *Some* (*uargs k*)) ∧
((π = *pwrite* ∧ *get-gs b* = *None* ∧ *msf* = {#}) ∨ (*pwrite* = *padd* π π′ ∧ *get-gs b* = *Some* (π′, *msf*)))
⟨*proof*⟩

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

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

lemma *denormalize-properties*:
shows *no-guard* (*denormalize H*)
and *full-ownership* (*get-fh* (*denormalize H*))
and *normalize* (*get-fh* (*denormalize H*)) = *H*
and *full-ownership* (*get-fh h*) ∧ *no-guard h* ⟹ *denormalize* (*normalize* (*get-fh h*)) = *h*
and *full-ownership* (*get-fh h*) ⟹ *denormalize* (*normalize* (*get-fh h*)) = *remove-guards h*
⟨*proof*⟩

lemma *no-guard-then-sat-star-uguard*:
assumes *no-guard h* ∧ *no-guard h*′
and (*s*, *h*), (*s*′, *h*′) ⊨ *Q*
shows (*s*, *add-uguard-to-no-guard k h* (*e s*)), (*s*′, *add-uguard-to-no-guard k h*′ (*e s*′)) ⊨ *Star Q* (*UniqueGuard k e*)
⟨*proof*⟩

lemma *no-guard-then-sat-star*:
assumes *no-guard h* ∧ *no-guard h*′
and (*s*, *h*), (*s*′, *h*′) ⊨ *Q*
shows (*s*, *add-sguard-to-no-guard h* π (*ms s*)), (*s*′, *add-sguard-to-no-guard h*′ π

$(ms\ s')) \models Star\ Q\ (SharedGuard\ \pi\ ms)$
⟨*proof*⟩

end

4.3 Safety and Hoare Triples

theory *Safety*
imports *Guards*
begin

4.3.1 Preliminaries

definition *sat-inv* :: *store* ⇒ ('*i*, '*a*) *heap* ⇒ ('*i*, '*a*, *nat*) *single-context* ⇒ *bool*
where
sat-inv s hj Γ ⟷ (*s*, *hj*), (*s*, *hj*) ⊨ *invariant* Γ ∧ *no-guard hj*

lemma *sat-invI*:
assumes (*s*, *hj*), (*s*, *hj*) ⊨ *invariant* Γ
and *no-guard hj*
shows *sat-inv s hj* Γ
⟨*proof*⟩

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* :: (*store* × ('*i*, '*a*) *heap*) *set* ⇒ *var set* ⇒ *bool* **where**
upper-fvs S vars ⟷ (∀ *s s' h*. (*s*, *h*) ∈ *S* ∧ *agrees vars s s'* ⟶ (*s'*, *h*) ∈ *S*)

Only need to agree on vars

definition *upperize* **where**
upperize S vars = { σ' |σ σ'. σ ∈ *S* ∧ *snd* σ = *snd* σ' ∧ *agrees vars* (*fst* σ) (*fst* σ')}

definition *close-var* **where**
close-var S x = { ((*fst* σ)(*x* := *v*), *snd* σ) |σ *v*. σ ∈ *S* }

lemma *upper-fvsI*:
assumes ⋀*s s' h*. (*s*, *h*) ∈ *S* ∧ *agrees vars s s'* ⟹ (*s'*, *h*) ∈ *S*
shows *upper-fvs S vars*
⟨*proof*⟩

lemma *pair-sat-comm*:
assumes *pair-sat S S' A*
shows *pair-sat S' S A*
⟨*proof*⟩

lemma *in-upperize*:
(*s'*, *h*) ∈ *upperize S vars* ⟷ (∃ *s*. (*s*, *h*) ∈ *S* ∧ *agrees vars s s'*) (**is** *?A* ⟷ *?B*)

⟨*proof*⟩

lemma *upper-fvs-upperize*:
  upper-fvs (*upperize S vars*) *vars*
⟨*proof*⟩

lemma *upperize-larger*:
  $S \subseteq$ *upperize S vars*
⟨*proof*⟩

lemma *pair-sat-upperize*:
  assumes *pair-sat S S′ A*
  shows *pair-sat* (*upperize S* (*fvA A*)) *S′ A*
⟨*proof*⟩

lemma *in-close-var*:
  $(s', h) \in$ *close-var S x* $\longleftrightarrow$ $(\exists\, s\ v.\ (s, h) \in S \wedge s' = s(x := v))$ (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

lemma *pair-sat-close-var*:
  assumes $x \notin$ *fvA A*
    and *pair-sat S S′ A*
  shows *pair-sat* (*close-var S x*) *S′ A*
⟨*proof*⟩

lemma *pair-sat-close-var-double*:
  assumes *pair-sat S S′ A*
    and $x \notin$ *fvA A*
  shows *pair-sat* (*close-var S x*) (*close-var S′ x*) *A*
  ⟨*proof*⟩

lemma *close-var-subset*:
  $S \subseteq$ *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* Γ
and (*h'* :: ('*i*, '*a*) *heap*) ≽ *h*
shows *view* Γ (*normalize* (*get-fh h*)) = *view* Γ (*normalize* (*get-fh h'*))
⟨*proof*⟩

4.3.2 Safety

fun *no-abort* :: ('*i*, '*a*, *nat*) *cont* ⇒ *cmd* ⇒ *store* ⇒ ('*i*, '*a*) *heap* ⇒ *bool* **where**
no-abort None C s h ⟷ (∀ *hf H*. *Some H* = *Some h* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *no-guard H*
⟶ ¬ *aborts C* (*s*, *normalize* (*get-fh H*)))
| *no-abort* (*Some* Γ) *C s h* ⟷ (∀ *hf H hj v0*. *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧
semi-consistent Γ *v0 H* ∧ *sat-inv s hj* Γ
⟶ ¬ *aborts C* (*s*, *normalize* (*get-fh H*)))

lemma *no-abortI*:
assumes ⋀(*hf* :: ('*i*, '*a*) *heap*) (*H* :: ('*i*, '*a*) *heap*). *Some H* = *Some h* ⊕ *Some hf* ∧ Δ = *None* ∧ *full-ownership* (*get-fh H*) ∧ *no-guard H* ⟹ ¬ *aborts C* (*s*, *normalize* (*get-fh H*))
and ⋀*H hf hj v0* Γ. Δ = *Some* Γ ∧ *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *semi-consistent* Γ *v0 H* ∧ *sat-inv s hj* Γ
⟹ ¬ *aborts C* (*s*, *normalize* (*get-fh H*))
shows *no-abort* Δ *C s* (*h* :: ('*i*, '*a*) *heap*)
⟨*proof*⟩

lemma *no-abortSomeI*:
assumes ⋀*H hf hj v0*. *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *semi-consistent* Γ *v0 H* ∧ *sat-inv s hj* Γ
⟹ ¬ *aborts C* (*s*, *normalize* (*get-fh H*))
shows *no-abort* (*Some* Γ) *C s* (*h* :: ('*i*, '*a*) *heap*)
⟨*proof*⟩

lemma *no-abortNoneI*:
assumes ⋀(*hf* :: ('*i*, '*a*) *heap*) (*H* :: ('*i*, '*a*) *heap*). *Some H* = *Some h* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *no-guard H* ⟹ ¬ *aborts C* (*s*, *normalize* (*get-fh H*))
shows *no-abort* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C s* (*h* :: ('*i*, '*a*) *heap*)
⟨*proof*⟩

lemma *no-abortE*:
assumes *no-abort* Δ *C s h*
shows *Some H* = *Some h* ⊕ *Some hf* ⟹ Δ = *None* ⟹ *full-ownership* (*get-fh H*) ⟹ *no-guard H* ⟹ ¬ *aborts C* (*s*, *normalize* (*get-fh H*))
and Δ = *Some* Γ ⟹ *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ⟹ *sat-inv s hj* Γ ⟹ *full-ownership* (*get-fh H*) ⟹ *semi-consistent* Γ *v0 H*

$\Longrightarrow \neg$ *aborts C* (*s*, *normalize* (*get-fh H*))
⟨*proof*⟩

fun *safe* :: *nat* ⇒ ('*i*, '*a*, *nat*) *cont* ⇒ *cmd* ⇒ (*store* × ('*i*, '*a*) *heap*) ⇒ (*store* × ('*i*, '*a*) *heap*) *set* ⇒ *bool* **where**
safe 0 - - - - ⟷ True

| *safe* (*Suc n*) *None C* (*s*, *h*) *S* ⟷ (*C* = *Cskip* ⟶ (*s*, *h*) ∈ *S*) ∧ *no-abort* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C s h* ∧
(∀ *H hf C' s' h'*. *Some H* = *Some h* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *no-guard H*
∧ *red C* (*s*, *normalize* (*get-fh H*)) *C'* (*s'*, *h'*)
⟶ (∃ *h'' H'*. *full-ownership* (*get-fh H'*) ∧ *no-guard H'* ∧ *h'* = *normalize* (*get-fh H'*) ∧ *Some H'* = *Some h''* ⊕ *Some hf* ∧ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C'* (*s'*, *h''*) *S*))

| *safe* (*Suc n*) (*Some* Γ) *C* (*s*, *h*) *S* ⟷ (*C* = *Cskip* ⟶ (*s*, *h*) ∈ *S*) ∧ *no-abort* (*Some* Γ) *C s h* ∧
(∀ *H hf C' s' h' hj v0*. *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *semi-consistent* Γ *v0 H* ∧ *sat-inv s hj* Γ
∧ *red C* (*s*, *normalize* (*get-fh H*)) *C'* (*s'*, *h'*)
⟶ (∃ *h'' H' hj'*. *full-ownership* (*get-fh H'*) ∧ *semi-consistent* Γ *v0 H'* ∧ *sat-inv s' hj'* Γ
∧ *h'* = *normalize* (*get-fh H'*) ∧ *Some H'* = *Some h''* ⊕ *Some hj'* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C'* (*s'*, *h''*) *S*))

lemma *safeNoneI*:
assumes *C* = *Cskip* ⟹ (*s*, *h*) ∈ *S*
and *no-abort None C s h*
and ⋀*H hf C' s' h'*. *Some H* = *Some h* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *no-guard H* ∧ *red C* (*s*, *normalize* (*get-fh H*)) *C'* (*s'*, *h'*)
⟹ (∃ *h'' H'*. *full-ownership* (*get-fh H'*) ∧ *no-guard H'* ∧ *h'* = *normalize* (*get-fh H'*) ∧ *Some H'* = *Some h''* ⊕ *Some hf* ∧ *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* ⟹ (*s*, *h*) ∈ *S*
and *no-abort* (*Some* Γ) *C s h*
and ⋀*H hf C' s' h' hj v0*. *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*)
∧ *semi-consistent* Γ *v0 H* ∧ *sat-inv s hj* Γ ∧ *red C* (*s*, *normalize* (*get-fh H*)) *C'* (*s'*, *h'*)
⟹ (∃ *h'' H' hj'*. *full-ownership* (*get-fh H'*) ∧ *semi-consistent* Γ *v0 H'* ∧ *sat-inv s' hj'* Γ
∧ *h'* = *normalize* (*get-fh H'*) ∧ *Some H'* = *Some h''* ⊕ *Some hj'* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C'* (*s'*, *h''*) *S*)
shows *safe* (*Suc n*) (*Some* Γ) *C* (*s*, *h* :: ('*i*, '*a*) *heap*) *S*

⟨*proof*⟩

lemma *safeI*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*
assumes *C* = *Cskip* ⟹ (*s*, *h*) ∈ *S*
and *no-abort* Δ *C s h*
and ⋀*H hf C' s' h'*. Δ = *None* ⟹ *Some H* = *Some h* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *no-guard H* ∧ *red C* (*s*, *normalize* (*get-fh H*)) *C'* (*s'*, *h'*)
⟹ (∃ *h'' H'*. *full-ownership* (*get-fh H'*) ∧ *no-guard H'* ∧ *h'* = *normalize* (*get-fh H'*) ∧ *Some H'* = *Some h''* ⊕ *Some hf* ∧ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C'* (*s'*, *h''*) *S*)
and ⋀*H hf C' s' h' hj v0* Γ. Δ = *Some* Γ ⟹ *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*)
∧ *semi-consistent* Γ *v0 H* ∧ *sat-inv s hj* Γ ∧ *red C* (*s*, *normalize* (*get-fh H*)) *C'* (*s'*, *h'*)
⟹ (∃ *h'' H' hj'*. *full-ownership* (*get-fh H'*) ∧ *semi-consistent* Γ *v0 H'* ∧ *sat-inv s' hj'* Γ
∧ *h'* = *normalize* (*get-fh H'*) ∧ *Some H'* = *Some h''* ⊕ *Some hj'* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C'* (*s'*, *h''*) *S*)
shows *safe* (*Suc n*) Δ *C* (*s*, *h* :: ('*i*, '*a*) *heap*) *S*
⟨*proof*⟩

lemma *safeSomeAltI*:
assumes *C* = *Cskip* ⟹ (*s*, *h*) ∈ *S*
and ⋀*H hf hj v0*. *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *semi-consistent* Γ *v0 H* ∧ *sat-inv s hj* Γ
⟹ ¬ *aborts C* (*s*, *normalize* (*get-fh H*))
and ⋀*H hf C' s' h' hj v0*. *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*)
∧ *semi-consistent* Γ *v0 H* ∧ *sat-inv s hj* Γ ⟹ *red C* (*s*, *normalize* (*get-fh H*)) *C'* (*s'*, *h'*)
⟹ (∃ *h'' H' hj'*. *full-ownership* (*get-fh H'*) ∧ *semi-consistent* Γ *v0 H'* ∧ *sat-inv s' hj'* Γ
∧ *h'* = *normalize* (*get-fh H'*) ∧ *Some H'* = *Some h''* ⊕ *Some hj'* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C'* (*s'*, *h''*) *S*)
shows *safe* (*Suc n*) (*Some* Γ) *C* (*s*, *h* :: ('*i*, '*a*) *heap*) *S*
⟨*proof*⟩

lemma *safeSomeE*:
assumes *safe* (*Suc n*) (*Some* Γ) *C* (*s*, *h* :: ('*i*, '*a*) *heap*) *S*
shows *C* = *Cskip* ⟹ (*s*, *h*) ∈ *S*
and *no-abort* (*Some* Γ) *C s h*
and *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ⟹ *full-ownership* (*get-fh H*)
⟹ *semi-consistent* Γ *v0 H* ⟹ *sat-inv s hj* Γ ⟹ *red C* (*s*, *normalize* (*get-fh H*)) *C'* (*s'*, *h'*)
⟹ (∃ *h'' H' hj'*. *full-ownership* (*get-fh H'*) ∧ *semi-consistent* Γ *v0 H'* ∧ *sat-inv*

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

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

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

4.3.3 Useful results about safety

lemma *no-abort-larger*:
assumes $h' \succeq h$
and $no\text{-}abort\ \Gamma\ C\ s\ h$
shows $no\text{-}abort\ \Gamma\ C\ s\ h'$
⟨*proof*⟩

lemma *safe-larger-set-aux*:
fixes $\Delta :: ('i,\ 'a,\ nat)\ cont$
assumes $safe\ n\ \Delta\ C\ (s,\ h)\ S$
and $S \subseteq S'$
shows $safe\ n\ \Delta\ C\ (s,\ h)\ S'$
⟨*proof*⟩

lemma *safe-larger-set*:
assumes $safe\ n\ \Delta\ C\ \sigma\ S$
and $S \subseteq S'$
shows $safe\ n\ \Delta\ C\ \sigma\ S'$
⟨*proof*⟩

lemma *safe-smaller-aux*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*
assumes $m \leq n$
and *safe n* Δ *C* (*s*, *h*) *S*
shows *safe m* Δ *C* (*s*, *h*) *S*
⟨*proof*⟩

lemma *safe-smaller*:
assumes $m \leq n$
and *safe n* Δ *C* σ *S*
shows *safe m* Δ *C* σ *S*
⟨*proof*⟩

lemma *safe-free-vars-aux*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*
assumes *safe n* Δ *C* (*s0*, *h*) *S*
and *agrees* (*fvC C* ∪ *vars*) *s0 s1*
and *upper-fvs S vars*
and $\bigwedge\Gamma.\ \Delta$ = *Some* $\Gamma \Longrightarrow$ *agrees* (*fvA* (*invariant* Γ)) *s0 s1*
shows *safe n* Δ *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* ∪ *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* Γ) *C* (*s*, *h*) *S*
and *agrees* (*fvC C* ∪ *vars* ∪ *fvA* (*invariant* Γ)) *s s*′
and *upper-fvs S vars*
shows *safe n* (*Some* Γ) *C* (*s*′, *h*) *S*
⟨*proof*⟩

lemma *safe-free-vars*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*
assumes *safe n* Δ *C* (*s*, *h*) *S*
and *agrees* (*fvC C* ∪ *vars*) *s s*′
and *upper-fvs S vars*
and $\bigwedge\Gamma.\ \Delta$ = *Some* $\Gamma \Longrightarrow$ *agrees* (*fvA* (*invariant* Γ)) *s s*′
shows *safe n* Δ *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 ⋀*s h n*. (*s*, *h*), (*s*, *h*) ⊨ *P* ⟹ *safe n* Γ *C* (*s*, *h*) (Σ (*s*, *h*))
and ⋀*s h s′ h′*. (*s*, *h*), (*s′*, *h′*) ⊨ *P* ⟹ *pair-sat* (Σ (*s*, *h*)) (Σ (*s′*, *h′*)) *Q*
shows *hoare-triple-valid* Γ *P C Q*
⟨*proof*⟩

lemma *hoare-triple-valid-smallerI*:
assumes ⋀σ *n*. σ, σ ⊨ *P* ⟹ *safe n* Γ *C* σ (Σ σ)
and ⋀σ σ′. σ, σ′ ⊨ *P* ⟹ *pair-sat* (Σ σ) (Σ σ′) *Q*
shows *hoare-triple-valid* Γ *P C Q*
⟨*proof*⟩

lemma *hoare-triple-validE*:
assumes *hoare-triple-valid* Γ *P C Q*
shows ∃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*)
⟨*proof*⟩

lemma *hoare-triple-valid-simplerE*:
assumes *hoare-triple-valid* Γ *P C Q*
and σ, σ′ ⊨ *P*
shows ∃ *S S′*. *safe n* Γ *C* σ *S* ∧ *safe n* Γ *C* σ′ *S′* ∧ *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*) ∈ *S*
shows *safe n* Δ *Cskip* (*s*, *h*) *S*
⟨*proof*⟩

theorem *rule-skip*:
hoare-triple-valid Γ *P Cskip 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 Δ :: (′*i*, ′*a*, *nat*) *cont*
assumes ⋀Γ. Δ = *Some* Γ ⟹ *x* ∉ *fvA* (*invariant* Γ)
shows *safe m* Δ (*Cassign x E*) (*s*, *h*) { (*s*(*x* := *edenot E s*), *h*) }
⟨*proof*⟩

theorem *assign-rule*:
fixes Δ :: (′*i*, ′*a*, *nat*) *cont*
assumes ⋀Γ. Δ = *Some* Γ ⟹ *x* ∉ *fvA* (*invariant* Γ)
and *collect-existentials P* ∩ *fvE 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*, (*Map.empty*, *gs*, *gu*)) { (*s*(*x* := *a*), (*Map.empty*(*a* ↦ (*pwrite*, *edenot E s*)), *gs*, *gu*)) |*a*. *True* }
⟨*proof*⟩

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

lemma *safe-new*:
fixes Δ :: (′*i*, ′*a*, *nat*) *cont*
assumes ⋀Γ. Δ = *Some* Γ ⟹ *x* ∉ *fvA* (*invariant* Γ) ∧ *view-function-of-inv* Γ
shows *safe n* Δ (*Calloc x E*) (*s*, (*Map.empty*, *gs*, *gu*)) { (*s*(*x* := *a*), (*Map.empty*(*a* ↦ (*pwrite*, *edenot E s*)), *gs*, *gu*)) |*a*. *True* }
⟨*proof*⟩

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

4.4.4 Write

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

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

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

lemma *safe-write*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*
assumes *fh* (*edenot loc s*) = *Some* (*pwrite*, *v*)
and $\bigwedge\Gamma.\ \Delta$ = *Some* $\Gamma \Longrightarrow$ *view-function-of-inv* Γ
shows *safe n* Δ (*Cwrite loc E*) (*s*, (*fh*, *gs*, *gu*)) { (*s*, (*fh*(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *gs*, *gu*)) }
⟨*proof*⟩

theorem *write-rule*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*
assumes $\bigwedge\Gamma.\ \Delta$ = *Some* $\Gamma \Longrightarrow$ *view-function-of-inv* Γ
and *v* $\notin$ *fvE loc*
shows *hoare-triple-valid* Δ (*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*) σ *C'* σ'
inductive-cases *aborts-read-cases*: *aborts* (*Cread x E*) σ

lemma *safe-read-None*:

safe n (*None* :: (*'i*, *'a*, *nat*) *cont*) (*Cread x E*) (*s*, ([*edenot E s* ↦ (*π*, *v*)], *gs*, *gu*)) { (*s*(*x* := *v*), ([*edenot E s* ↦ (*π*, *v*)], *gs*, *gu*)) }
⟨*proof*⟩

lemma *safe-read-Some*:
assumes *view-function-of-inv* Γ
and *x* ∉ *fvA* (*invariant* Γ)
shows *safe n* (*Some* Γ) (*Cread x E*) (*s*, ([*edenot E s* ↦ (*π*, *v*)], *gs*, *gu*)) { (*s*(*x* := *v*), ([*edenot E s* ↦ (*π*, *v*)], *gs*, *gu*)) }
⟨*proof*⟩

lemma *safe-read*:
fixes Δ :: (*'i*, *'a*, *nat*) *cont*
assumes ⋀Γ. Δ = *Some* Γ ⟹ *x* ∉ *fvA* (*invariant* Γ) ∧ *view-function-of-inv* Γ
shows *safe n* Δ (*Cread x E*) (*s*, ([*edenot E s* ↦ (*π*, *v*)], *gs*, *gu*)) { (*s*(*x* := *v*), ([*edenot E s* ↦ (*π*, *v*)], *gs*, *gu*)) }
⟨*proof*⟩

theorem *read-rule*:
fixes Δ :: (*'i*, *'a*, *nat*) *cont*
assumes ⋀Γ. Δ = *Some* Γ ⟹ *x* ∉ *fvA* (*invariant* Γ) ∧ *view-function-of-inv* Γ
and *x* ∉ *fvE E* ∪ *fvE e*
shows *hoare-triple-valid* Δ (*PointsTo E π e*) (*Cread x E*) (*And* (*PointsTo E π e*) (*Bool* (*Beq* (*Evar x*) *e*)))
⟨*proof*⟩

4.4.6 Share

lemma *share-no-abort*:
assumes *no-abort* (*Some* Γ) *C s* (*h* :: (*'i*, *'a*) *heap*)
and *Some* (*h'* :: (*'i*, *'a*) *heap*) = *Some h* ⊕ *Some hj*
and *sat-inv s hj* Γ
and *get-gs h* = *Some* (*pwrite*, *sargs*)
and ⋀*k*. *get-gu h k* = *Some* (*uargs k*)
and *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs uargs* (*view* Γ (*normalize* (*get-fh hj*)))
and *view-function-of-inv* Γ
shows *no-abort None C s* (*remove-guards h'*)
⟨*proof*⟩

definition *S-after-share* **where**
S-after-share S Γ *v0* = { (*s*, *remove-guards h'*) |*h hj h' s*. *semi-consistent* Γ *v0 h'* ∧ *Some h'* = *Some h* ⊕ *Some hj* ∧ (*s*, *h*) ∈ *S* ∧ *sat-inv s hj* Γ }

lemma *share-lemma*:
assumes *safe n* (*Some* Γ) *C* (*s*, *h* :: (*'i*, *'a*) *heap*) *S*
and *Some* (*h'* :: (*'i*, *'a*) *heap*) = *Some h* ⊕ *Some hj*
and *sat-inv s hj* Γ
and *semi-consistent* Γ *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* Γ *v0*)
⟨*proof*⟩

definition *no-need-guards* **where**
no-need-guards A ⟷ (∀ *s1 h1 s2 h2*. (*s1*, *h1*), (*s2*, *h2*) ⊨ *A* ⟶ (*s1*, *remove-guards h1*), (*s2*, *remove-guards h2*) ⊨ *A*)

lemma *has-guard-then-safe-none*:
assumes ¬ *no-guard h*
and *C* = *Cskip* ⟹ (*s*, *h*) ∈ *S*
shows *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C* (*s*, *h*) *S*
⟨*proof*⟩

theorem *share-rule*:
fixes Γ :: ('*i*, '*a*, *nat*) *single-context*
assumes Γ = (| *view* = *f*, *abstract-view* = α, *saction* = *sact*, *uaction* = *uact*, *invariant* = *J* |)
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* ∧ *precise J*
and *wf-indexed-precondition upre* ∧ *wf-precondition spre*
and *x* ∉ *fvA J*
and *no-guard-assertion* (*Star P* (*LowView* (α ∘ *f*) *J x*))
shows *hoare-triple-valid* (*None* :: ('*i*, '*a*, *nat*) *cont*) (*Star P* (*LowView* (α ∘ *f*) *J x*)) *C* (*Star Q* (*LowView* (α ∘ *f*) *J x*))
⟨*proof*⟩

4.4.7 Atomic

lemma *red-rtrans-induct*:
assumes *red-rtrans C* σ *C'* σ'
and ⋀*C* σ. *P C* σ *C* σ
and ⋀*C* σ *C'* σ' *C''* σ''. *red C* σ *C'* σ' ⟹ *red-rtrans C'* σ' *C''* σ'' ⟹ *P C'* σ' *C''* σ'' ⟹ *P C* σ *C''* σ''
shows *P C* σ *C'* σ'
⟨*proof*⟩

lemma *safe-atomic*:
assumes *red-rtrans C1* σ*1 C2* σ*2*
and σ*1* = (*s1*, *H1*)
and σ*2* = (*s2*, *H2*)

and $\bigwedge n.$ *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C1* (*s1*, *h*) *S*
and *H* = *denormalize H1*
and *Some H* = *Some h* ⊕ *Some hf*
and *full-ownership* (*get-fh H*) ∧ *no-guard H*
shows ¬ *aborts C2 σ2* ∧ (*C2* = *Cskip* ⟶
(∃ *h1 H'*. *Some H'* = *Some h1* ⊕ *Some hf* ∧ *H2* = *normalize* (*get-fh* (*H'*)) ∧ *no-guard H'* ∧ *full-ownership* (*get-fh H'*) ∧ (*s2*, *h1*) ∈ *S*))
⟨*proof*⟩

theorem *atomic-rule-unique*:
fixes Γ :: (*'i*, *'a*, *nat*) *single-context*

fixes *map-to-list* :: *nat* ⇒ *'a list*
fixes *map-to-arg* :: *nat* ⇒ *'a*

assumes Γ = (| *view* = *f*, *abstract-view* = α, *saction* = *sact*, *uaction* = *uact*, *invariant* = *J* |)
and *hoare-triple-valid* (*None* :: (*'i*, *'a*, *nat*) *cont*) (*Star P* (*View f J* (λ*s*. *s x*)))
C (*Star Q* (*View f J* (λ*s*. *uact index* (*s x*) (*map-to-arg* (*s uarg*)))))

and *precise J* ∧ *unary J*
and *view-function-of-inv* Γ
and *x* ∉ *fvC C* ∪ *fvA P* ∪ *fvA Q* ∪ *fvA J*

and *uarg* ∉ *fvC C*
and *l* ∉ *fvC C*

and *x* ∉ *fvS* (λ*s*. *map-to-list* (*s l*))
and *x* ∉ *fvS* (λ*s*. *map-to-arg* (*s uarg*) # *map-to-list* (*s l*))

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

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

theorem *atomic-rule-shared*:
fixes Γ :: (*'i*, *'a*, *nat*) *single-context*

fixes *map-to-multiset* :: *nat* ⇒ *'a multiset*
fixes *map-to-arg* :: *nat* ⇒ *'a*

assumes Γ = (| *view* = *f*, *abstract-view* = α, *saction* = *sact*, *uaction* = *uact*, *invariant* = *J* |)
and *hoare-triple-valid* (*None* :: (*'i*, *'a*, *nat*) *cont*) (*Star P* (*View f J* (λ*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* Γ
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* Γ) (*Star* P (*SharedGuard* π ($\lambda s.$ *map-to-multiset* (s *ms*)))) (*Catomic* C)
(*Star* Q (*SharedGuard* π ($\lambda s.$ {# *map-to-arg* (s *sarg*) #} + *map-to-multiset* (s *ms*))))
⟨*proof*⟩

4.4.8 Parallel

lemma *par-cases*:
assumes *red* (*Cpar C1 C2*) σ C' σ'
and $\bigwedge$*C1'*. $C' =$ *Cpar C1'* *C2* $\wedge$ *red C1* σ *C1'* $\sigma' \Longrightarrow P$
and $\bigwedge$*C2'*. $C' =$ *Cpar C1* *C2'* $\wedge$ *red C2* σ *C2'* $\sigma' \Longrightarrow P$
and *C1* $=$ *Cskip* $\wedge$ *C2* $=$ *Cskip* $\wedge$ $C' =$ *Cskip* $\wedge$ $\sigma = \sigma' \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *no-abort-par*:
assumes *no-abort* Γ *C1* s h
and *no-abort* Γ *C2* s h
shows *no-abort* Γ (*Cpar C1 C2*) s h
⟨*proof*⟩

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

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

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

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* = *Some h1* ⊕ *Some h2*

and *disjoint* (*fvC C1* ∪ *vars1*) (*wrC C2*)
and *disjoint* (*fvC C2* ∪ *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 Δ :: ('*i*, '*a*, *nat*) *cont*
assumes *safe n* Δ *C1* (*s*, *h1*) *S1*
and *safe n* Δ *C2* (*s*, *h2*) *S2*
and *Some h* = *Some h1* ⊕ *Some h2*
and *disjoint* (*fvC C1* ∪ *vars1*) (*wrC C2*)
and *disjoint* (*fvC C2* ∪ *vars2*) (*wrC C1*)
and *upper-fvs S1 vars1*
and *upper-fvs S2 vars2*

and ⋀Γ. Δ = *Some* Γ ⟹ *disjoint* (*fvA* (*invariant* Γ)) (*wrC C2*)
and ⋀Γ. Δ = *Some* Γ ⟹ *disjoint* (*fvA* (*invariant* Γ)) (*wrC C1*)

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

theorem *rule-par*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*

assumes *hoare-triple-valid* Δ *P1 C1 Q1*
and *hoare-triple-valid* Δ *P2 C2 Q2*
and *disjoint* (*fvA P1* ∪ *fvC C1* ∪ *fvA Q1*) (*wrC C2*)
and *disjoint* (*fvA P2* ∪ *fvC C2* ∪ *fvA Q2*) (*wrC C1*)

and ⋀Γ. Δ = *Some* Γ ⟹ *disjoint* (*fvA* (*invariant* Γ)) (*wrC C2*)
and ⋀Γ. Δ = *Some* Γ ⟹ *disjoint* (*fvA* (*invariant* Γ)) (*wrC C1*)

and *precise P1* ∨ *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* ⟹ *s* = *s'* ∧ *h* = *h'* ⟹ *bdenot b s* ⟹ *P*
and *C'* = *C2* ⟹ *s* = *s'* ∧ *h* = *h'* ⟹ ¬ *bdenot b s* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *if-safe-None*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*

assumes *bdenot b s* ⟹ *safe n* Δ *C1* (*s*, *h*) *S*
and ¬ *bdenot b s* ⟹ *safe n* Δ *C2* (*s*, *h*) *S*
and Δ = *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* ⟹ *safe n* (*Some* Γ) *C1* (*s*, *h*) *S*
and ¬ *bdenot b s* ⟹ *safe n* (*Some* Γ) *C2* (*s*, *h*) *S*
shows *safe* (*Suc n*) (*Some* Γ) (*Cif b C1 C2*) (*s*, *h*) *S*
⟨*proof*⟩

lemma *if-safe*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*
assumes *bdenot b s* ⟹ *safe n* Δ *C1* (*s*, *h*) *S*
and ¬ *bdenot b s* ⟹ *safe n* Δ *C2* (*s*, *h*) *S*
shows *safe* (*Suc n*) Δ (*Cif b C1 C2*) (*s*, *h*) *S*
⟨*proof*⟩

theorem *if1-rule*:
fixes Δ :: ('*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 Δ :: ('*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* Δ *P* (*Cif b C1 C2*) *Q*
⟨*proof*⟩

4.4.10 Sequential composition

inductive-cases *red-seq-cases*: *red* (*Cseq C1 C2*) σ *C'* σ'

lemma *aborts-seq-aborts-C1*:
assumes *aborts* (*Cseq C1 C2*) σ
shows *aborts C1* σ
⟨*proof*⟩

lemma *safe-seq-None*:
assumes *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C1* (*s*, *h*) *S1*
and $\bigwedge m\ s'\ h'.\ m \leq n \wedge (s', h') \in S1 \Longrightarrow$ *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*
⟨*proof*⟩

lemma *safe-seq-Some*:
assumes *safe n* (*Some* Γ) *C1* (*s*, *h*) *S1*
and $\bigwedge m\ s'\ h'.\ m \leq n \wedge (s', h') \in S1 \Longrightarrow$ *safe m* (*Some* Γ) *C2* (*s'*, *h'*) *S2*
shows *safe n* (*Some* Γ) (*Cseq C1 C2*) (*s*, *h*) *S2*
⟨*proof*⟩

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

theorem *seq-rule*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*
assumes *hoare-triple-valid* Δ *P C1 R*
and *hoare-triple-valid* Δ *R C2 Q*
shows *hoare-triple-valid* Δ *P* (*Cseq C1 C2*) *Q*
⟨*proof*⟩

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''*})
⟨*proof*⟩

lemma *safe-frame-Some*:

assumes *safe n* (*Some* Γ) *C* (*s*, *h*) *S*
and *Some H* = *Some h* ⊕ *Some hf0*
shows *safe n* (*Some* Γ) *C* (*s*, *H*) (*add-states S* {(*s''*, *hf0*) |*s''*. *agrees* (− *wrC C*) *s s''*})
⟨*proof*⟩

lemma *safe-frame*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*
assumes *safe n* Δ *C* (*s*, *h*) *S*
and *Some H* = *Some h* ⊕ *Some hf0*
shows *safe n* Δ *C* (*s*, *H*) (*add-states S* {(*s''*, *hf0*) |*s''*. *agrees* (− *wrC C*) *s s''*})
⟨*proof*⟩

theorem *frame-rule*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*
assumes *hoare-triple-valid* Δ *P C Q*
and *disjoint* (*fvA R*) (*wrC C*)
and *precise P* ∨ *precise R*
shows *hoare-triple-valid* Δ (*Star P R*) *C* (*Star Q R*)
⟨*proof*⟩

4.4.12 Consequence

theorem *consequence-rule*:
fixes Δ :: ('*i*, '*a*, *nat*) *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 Δ :: ('*i*, '*a*, *nat*) *cont*
assumes *hoare-triple-valid* Δ *P C Q*
and *x* ∉ *fvC C*
and ⋀Γ. Δ = *Some* Γ ⟹ *x* ∉ *fvA* (*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
| ⟦ *leads-to-loop b I* Σ σ σ' ; *bdenot b* (*fst* σ') ; σ'' ∈ Σ σ' ⟧ ⟹ *leads-to-loop b I* Σ σ σ''

definition *leads-to-loop-set* **where**
leads-to-loop-set b I Σ σ = { *σ′* |*σ′*. *leads-to-loop b I Σ σ σ′*}

definition *trans-Σ* **where**
trans-Σ b I Σ σ = *Set.filter* (λ*σ*. ¬ *bdenot b* (*fst σ*)) (*leads-to-loop-set b I Σ σ*)

inductive-cases *red-while-cases*: *red* (*Cwhile b s*) *σ C′ σ′*
inductive-cases *abort-while-cases*: *aborts* (*Cwhile b s*) *σ*

lemma *safe-while-None*:
assumes ⋀*σ m*. *σ*, *σ* ⊨ *And I* (*Bool b*) ⟹ *safe n* (*None* :: (*′i*, *′a*, *nat*) *cont*) *C* *σ* (*Σ σ*)
and ⋀*σ σ′*. *σ*, *σ′* ⊨ *And I* (*Bool b*) ⟹ *pair-sat* (*Σ σ*) (*Σ σ′*) *I*
and (*s*, *h*), (*s*, *h*) ⊨ *I*
and *leads-to-loop b I Σ σ* (*s*, *h*)
shows *safe n* (*None* :: (*′i*, *′a*, *nat*) *cont*) (*Cwhile b C*) (*s*, *h*) (*trans-Σ b I Σ σ*)
⟨*proof*⟩

lemma *safe-while-Some*:
assumes ⋀*σ m*. *σ*, *σ* ⊨ *And I* (*Bool b*) ⟹ *safe n* (*Some* Γ) *C σ* (*Σ σ*)
and ⋀*σ σ′*. *σ*, *σ′* ⊨ *And I* (*Bool b*) ⟹ *pair-sat* (*Σ σ*) (*Σ σ′*) *I*
and (*s*, *h*), (*s*, *h*) ⊨ *I*
and *leads-to-loop b I Σ σ* (*s*, *h*)
shows *safe n* (*Some* Γ) (*Cwhile b C*) (*s*, *h*) (*trans-Σ b I Σ σ*)
⟨*proof*⟩

lemma *safe-while*:
fixes Δ :: (*′i*, *′a*, *nat*) *cont*
assumes ⋀*σ m*. *σ*, *σ* ⊨ *And I* (*Bool b*) ⟹ *safe n* Δ *C σ* (*Σ σ*)
and ⋀*σ σ′*. *σ*, *σ′* ⊨ *And I* (*Bool b*) ⟹ *pair-sat* (*Σ σ*) (*Σ σ′*) *I*
and (*s*, *h*), (*s*, *h*) ⊨ *I*
and *leads-to-loop b I Σ σ* (*s*, *h*)
shows *safe n* Δ (*Cwhile b C*) (*s*, *h*) (*trans-Σ b I Σ σ*)
⟨*proof*⟩

lemma *leads-to-sat-inv-unary*:
assumes *leads-to-loop b I Σ σ σ′*
and ⋀*σ σ′*. *σ*, *σ′* ⊨ (*And I* (*Bool b*)) ⟹ *pair-sat* (*Σ σ*) (*Σ σ′*) *I*
and *σ*, *σ* ⊨ *I*
shows *σ′*, *σ′* ⊨ *I*
⟨*proof*⟩

theorem *while-rule2*:
fixes Δ :: (*′i*, *′a*, *nat*) *cont*
assumes *unary I*
and *hoare-triple-valid* Δ (*And I* (*Bool b*)) *C I*
shows *hoare-triple-valid* Δ *I* (*Cwhile b C*) (*And I* (*Bool* (*Bnot b*)))
⟨*proof*⟩

fun *iterate-sigma* :: *nat* ⇒ *bexp* ⇒ ('*i*, '*a*, *nat*) *assertion* ⇒ ((*store* × ('*i*, '*a*) *heap*) ⇒ (*store* × ('*i*, '*a*) *heap*) *set*) ⇒ (*store* × ('*i*, '*a*) *heap*) ⇒ (*store* × ('*i*, '*a*) *heap*) *set*
where
iterate-sigma 0 b I Σ σ = {*σ*}
| *iterate-sigma* (*Suc n*) *b I Σ σ* = (⋃*σ'* ∈ *Set.filter* (λ*σ*. *bdenot b* (*fst σ*)) (*iterate-sigma n b I Σ σ*). *Σ σ'*)

lemma *union-of-iterate-sigma-is-leads-to-loop-set*:
assumes *leads-to-loop b I Σ σ σ'*
shows *σ'* ∈ (⋃ *n*. *iterate-sigma n b I Σ σ*)
⟨*proof*⟩

lemma *trans-included*:
trans-Σ b I Σ σ ⊆ *Set.filter* (λ*σ*. ¬ *bdenot b* (*fst σ*)) (⋃ *n*. *iterate-sigma n b I Σ σ*)
⟨*proof*⟩

lemma *iterate-sigma-low-all-sat-I-and-low*:
assumes ⋀*σ σ'*. *σ*, *σ'* ⊨ (*And I* (*Bool b*)) ⟹ *pair-sat* (*Σ σ*) (*Σ σ'*) (*And I* (*Low b*))
and *σ1*, *σ2* ⊨ *I*
and *bdenot b* (*fst σ1*) = *bdenot b* (*fst σ2*)
shows *pair-sat* (*iterate-sigma n b I Σ σ1*) (*iterate-sigma n b I Σ σ2*) (*And I* (*Low b*))
⟨*proof*⟩

lemma *iterate-empty-later-empty*:
assumes *iterate-sigma n b I Σ σ* = {}
and *m* ≥ *n*
shows *iterate-sigma m b I Σ σ* = {}
⟨*proof*⟩

lemma *all-same*:
assumes ⋀*σ σ'*. *σ*, *σ'* ⊨ (*And I* (*Bool b*)) ⟹ *pair-sat* (*Σ σ*) (*Σ σ'*) (*And I* (*Low b*))
and *σ1*, *σ2* ⊨ *I*
and *bdenot b* (*fst σ1*) = *bdenot b* (*fst σ2*)
and *x1* ∈ *iterate-sigma n b I Σ σ1*
and *x2* ∈ *iterate-sigma n b I Σ σ2*
shows *bdenot b* (*fst x1*) = *bdenot b* (*fst x2*)
⟨*proof*⟩

lemma *non-empty-at-most-once*:
assumes ⋀*σ σ'*. *σ*, *σ'* ⊨ (*And I* (*Bool b*)) ⟹ *pair-sat* (*Σ σ*) (*Σ σ'*) (*And I* (*Low b*))

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

lemma *one-non-empty-union*:
assumes $\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models (And\ I\ (Bool\ b)) \Longrightarrow pair\text{-}sat\ (\Sigma\ \sigma)\ (\Sigma\ \sigma')\ (And\ I\ (Low\ b))$
and σ, $\sigma \models I$
and *Set.filter* $(\lambda\sigma.\ \neg\ bdenot\ b\ (fst\ \sigma))$ $(iterate\text{-}sigma\ k\ b\ I\ \Sigma\ \sigma) \neq \{\}$
shows *Set.filter* $(\lambda\sigma.\ \neg\ bdenot\ b\ (fst\ \sigma))$ $(\bigcup n.\ iterate\text{-}sigma\ n\ b\ I\ \Sigma\ \sigma) =$ *Set.filter* $(\lambda\sigma.\ \neg\ bdenot\ b\ (fst\ \sigma))$ $(iterate\text{-}sigma\ k\ b\ I\ \Sigma\ \sigma)$
⟨*proof*⟩

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

lemma *union-exists-at-some-point-exactly*:
assumes $\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models (And\ I\ (Bool\ b)) \Longrightarrow pair\text{-}sat\ (\Sigma\ \sigma)\ (\Sigma\ \sigma')\ (And\ I\ (Low\ b))$
and $\sigma 1$, $\sigma 2 \models I$
and $bdenot\ b\ (fst\ \sigma 1) = bdenot\ b\ (fst\ \sigma 2)$
and *Set.filter* $(\lambda\sigma.\ \neg\ bdenot\ b\ (fst\ \sigma))$ $(\bigcup n.\ iterate\text{-}sigma\ n\ b\ I\ \Sigma\ \sigma 1) \neq \{\}$
and *Set.filter* $(\lambda\sigma.\ \neg\ bdenot\ b\ (fst\ \sigma))$ $(\bigcup n.\ iterate\text{-}sigma\ n\ b\ I\ \Sigma\ \sigma 2) \neq \{\}$
shows $\exists k.\ not\text{-}set\ b\ (\bigcup n.\ iterate\text{-}sigma\ n\ b\ I\ \Sigma\ \sigma 1) = not\text{-}set\ b\ (iterate\text{-}sigma\ k\ b\ I\ \Sigma\ \sigma 1) \wedge not\text{-}set\ b\ (\bigcup n.\ iterate\text{-}sigma\ n\ b\ I\ \Sigma\ \sigma 2) = not\text{-}set\ b\ (iterate\text{-}sigma\ k\ b\ I\ \Sigma\ \sigma 2)$
⟨*proof*⟩

theorem *while-rule1*:
fixes $\Delta :: ('i, 'a, nat)\ cont$
assumes *hoare-triple-valid* $\Delta\ (And\ I\ (Bool\ b))\ C\ (And\ I\ (Low\ b))$
shows *hoare-triple-valid* $\Delta\ (And\ I\ (Low\ b))\ (Cwhile\ b\ C)\ (And\ I\ (Bool\ (Bnot\ b)))$
⟨*proof*⟩

lemma *entails-smallerI*:
assumes $\bigwedge s1\ h1\ s2\ h2.\ (s1, h1), (s2, h2) \models A \Longrightarrow (s1, h1), (s2, h2) \models B$
shows *entails* $A\ B$
⟨*proof*⟩

corollary *while-rule*:
fixes $\Delta :: ('i, 'a, nat)\ cont$
assumes *entails* $P\ (Star\ P'\ R)$
and *unary* P'

and *fvA R ∩ wrC C = {}*
and *hoare-triple-valid Δ (And P' (Bool e)) C P'*
and *hoare-triple-valid Δ (And P (Bool (Band e e'))) C (And P (Low (Band e e')))*
and *precise P' ∨ precise R*
shows *hoare-triple-valid Δ (And P (Low (Band e e'))) (Cseq (Cwhile (Band e e') C) (Cwhile e C)) (And (Star P' R) (Bool (Bnot e)))*
⟨*proof*⟩

4.4.15 CommCSL is sound

theorem *soundness*:
assumes *Δ ⊢ {P} C {Q}*
shows *Δ ⊨ {P} C {Q}*
⟨*proof*⟩

4.5 Corollaries

theorem *safety*:
assumes *hoare-triple-valid (None :: ('i, 'a, nat) cont) P C Q*
and *(s1, h1), (s2, h2) ⊨ P*

and *Some H1 = Some h1 ⊕ Some hf1 ∧ full-ownership (get-fh H1) ∧ no-guard H1*
— extend h1 to a normal state H1 without guards

and *Some H2 = Some h2 ⊕ Some hf2 ∧ full-ownership (get-fh H2) ∧ no-guard H2*
— extend h2 to a normal state H2 without guards

shows *⋀σ' C'. red-rtrans C (s1, normalize (get-fh H1)) C' σ' ⟹ ¬ aborts C' σ'*
and *⋀σ' C'. red-rtrans C (s2, normalize (get-fh H2)) C' σ' ⟹ ¬ aborts C' σ'*
and *⋀σ1' σ2'. red-rtrans C (s1, normalize (get-fh H1)) Cskip σ1'*
⟹ red-rtrans C (s2, normalize (get-fh H2)) Cskip σ2'
⟹ (∃h1' h2' H1' H2'. no-guard H1' ∧ full-ownership (get-fh H1') ∧ snd σ1' = normalize (get-fh H1') ∧ Some H1' = Some h1' ⊕ Some hf1
∧ no-guard H2' ∧ full-ownership (get-fh H2') ∧ snd σ2' = normalize (get-fh H2') ∧ Some H2' = Some h2' ⊕ Some hf2
∧ (fst σ1', h1'), (fst σ2', h2') ⊨ Q)
⟨*proof*⟩

lemma *neutral-add*:
Some h = Some h ⊕ Some (Map.empty, None, (λ-. None))
⟨*proof*⟩

corollary *safety-no-frame*:
assumes *hoare-triple-valid (None :: ('i, 'a, nat) cont) P C Q*
and *(s1, H1), (s2, H2) ⊨ P*

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

shows ⋀σ′ *C*′. *red-rtrans C* (*s1*, *normalize* (*get-fh H1*)) *C*′ σ′ ⟹ ¬ *aborts C*′ σ′
and ⋀σ′ *C*′. *red-rtrans C* (*s2*, *normalize* (*get-fh H2*)) *C*′ σ′ ⟹ ¬ *aborts C*′ σ′
and ⋀σ1′ σ2′. *red-rtrans C* (*s1*, *normalize* (*get-fh H1*)) *Cskip* σ1′
⟹ *red-rtrans C* (*s2*, *normalize* (*get-fh H2*)) *Cskip* σ2′
⟹ (∃ *H1*′ *H2*′. *no-guard H1*′ ∧ *full-ownership* (*get-fh H1*′) ∧ *snd* σ1′ = *normalize* (*get-fh H1*′)
∧ *no-guard H2*′ ∧ *full-ownership* (*get-fh H2*′) ∧ *snd* σ2′ = *normalize* (*get-fh H2*′)
∧ (*fst* σ1′, *H1*′), (*fst* σ2′, *H2*′) ⊨ *Q*)
⟨*proof*⟩

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*) ⊨ *low-list l*
and *x* ∈ *set l*
shows *s1 x* = *s2 x*
⟨*proof*⟩

lemma *low-listI*:
assumes ⋀*x*. *x* ∈ *set l* ⟹ *s1 x* = *s2 x*
shows (*s1*, *h1*), (*s2*, *h2*) ⊨ *low-list l*
⟨*proof*⟩

corollary *non-interference*:
assumes (*None* :: ('*i*, '*a*, *nat*) *cont*) ⊢ {*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 ⋀*x*. *x* ∈ *set In* ⟹ *s1 x* = *s2 x*
and *x* ∈ *set Out*
and (*s1*, *H1*), (*s2*, *H2*) ⊨ *P*
and *full-ownership* (*get-fh H1*) ∧ *no-guard H1*
and *full-ownership* (*get-fh H2*) ∧ *no-guard H2*
shows *s1*′ *x* = *s2*′ *x*
⟨*proof*⟩

definition *heapify* **where**
heapify h = (λ*l*. *apply-opt* (λ*v*. (*pwrite*, *v*)) (*h l*), *None*, λ-. *None*)

lemma *heapify-properties*:
full-ownership (*get-fh* (*heapify h*))
no-guard (*heapify h*)
normalize (*get-fh* (*heapify h*)) = *h*
⟨*proof*⟩

corollary *non-interference-no-precondition*:
assumes (*None* :: ('*i*, '*a*, *nat*) *cont*) ⊢ {*low-list In*} *C* {*low-list Out*}
and *red-rtrans C* (*s1*, *h1*) *Cskip* (*s1'*, *h1'*)
and *red-rtrans C* (*s2*, *h2*) *Cskip* (*s2'*, *h2'*)
and ⋀*x*. *x* ∈ *set In* ⟹ *s1 x* = *s2 x*
and *x* ∈ *set Out*
shows *s1' x* = *s2' x*
⟨*proof*⟩

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.