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

March 17, 2025

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

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

theory *PartialMap*
  imports *Main*
begin

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

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* $(a\ p)\ (b\ p)$

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

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

lemma *compatible-mapsI*:
  assumes $\bigwedge x\ a\ b.\ h1\ x = Some\ a \wedge h2\ x = Some\ b \Longrightarrow f\ a\ b$
  shows *compatible-maps f h1 h2*
  by (*metis assms compatible-maps-def compatible-options.elims*(*3*))

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*
  by (*metis assms map-included-def option.exhaust*)

lemma *compatible-maps-empty*:
  compatible-maps f h (*Map.empty*)
  by (*simp add*: *compatible-maps-def*)

lemma *compatible-maps-comm*:
compatible-maps $(=)$ *h1 h2* $\longleftrightarrow$ *compatible-maps* $(=)$ *h2 h1*
proof $-$
have $\bigwedge a\ b.$ *compatible-maps* $(=)$ *a b* $\Longrightarrow$ *compatible-maps* $(=)$ *b a*
by (*metis* (*mono-tags*, *lifting*) *compatible-mapsI compatible-maps-def compatible-options.simps*(*1*))
then show *?thesis*
by *auto*
qed

lemma *add-heaps-asso*:
(*h1* ++ *h2*) ++ *h3* = *h1* ++ (*h2* ++ *h3*)
by *auto*

lemma *compatible-maps-same*:
assumes *compatible-maps* $(=)$ *ha hb*
and *ha x* = *Some y*
shows (*ha* ++ *hb*) *x* = *Some y*
proof (*cases hb x*)
case *None*
then show *?thesis*
by (*simp add*: *assms*(*2*) *map-add-Some-iff*)
next
case (*Some a*)
then show *?thesis*
by (*metis* (*mono-tags*) *assms*(*1*) *assms*(*2*) *compatible-maps-def compatible-options.simps*(*1*) *map-add-def option.simps*(*5*))
qed

lemma *compatible-maps-refl*:
compatible-maps $(=)$ *h h*
using *compatible-maps-def compatible-options.elims*(*3*) **by** *fastforce*

lemma *map-invo*:
h ++ *h* = *h*
by (*simp add*: *map-add-subsumed2*)

lemma *included-then-compatible-maps*:
assumes *map-included h1 h*
and *map-included h2 h*
shows *compatible-maps* $(=)$ *h1 h2*
proof (*rule compatible-mapsI*)
fix *x a b* **assume** *h1 x* = *Some a* $\wedge$ *h2 x* = *Some b*
show *a* = *b*
by (*metis* ‹*h1 x* = *Some a* $\wedge$ *h2 x* = *Some b*› *assms*(*1*) *assms*(*2*) *map-included-def option.inject option.simps*(*3*))
qed

lemma *commut-charact*:
  assumes *compatible-maps* (=) *h1 h2*
  shows *h1* ++ *h2* = *h2* ++ *h1*
proof (*rule ext*)
  fix *x*
  show (*h1* ++ *h2*) *x* = (*h2* ++ *h1*) *x*
  proof (*cases h1 x*)
    case *None*
    then show *?thesis*
     by (*simp add*: *domIff map-add-dom-app-simps*(*2*) *map-add-dom-app-simps*(*3*))
  next
    case (*Some a*)
    then show *?thesis*
      by (*simp add*: *assms compatible-maps-same*)
  qed
qed

end

1.2 Fractional Permissions

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

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

typedef *prat* = { *r* :: *rat* |*r*. *r* > *0*} **by** *fastforce*

setup-lifting *type-definition-prat*

lift-definition *pwrite* :: *prat* **is** *1* **by** *simp*
lift-definition *half* :: *prat* **is** *1* / *2* **by** *fastforce*

lift-definition *pgte* :: *prat* ⇒ *prat* ⇒ *bool* **is** (≥) **done**
lift-definition *pgt* :: *prat* ⇒ *prat* ⇒ *bool* **is** (>) **done**
lift-definition *lt* :: *prat* ⇒ *prat* ⇒ *bool* **is** (<) **done**

lift-definition *pmult* :: *prat* ⇒ *prat* ⇒ *prat* **is** (∗) **by** *simp*
lift-definition *padd* :: *prat* ⇒ *prat* ⇒ *prat* **is** (+) **by** *simp*

lift-definition *pdiv* :: *prat* ⇒ *prat* ⇒ *prat* **is** (/) **by** *simp*

lift-definition *pmin* :: *prat* ⇒ *prat* ⇒ *prat* **is** (*min*) **by** *simp*
lift-definition *pmax* :: *prat* ⇒ *prat* ⇒ *prat* **is** (*max*) **by** *simp*

lemma *pmin-comm*:
  pmin a b = *pmin b a*
  by (*metis Rep-prat-inverse min.commute pmin.rep-eq*)

lemma *pmin-greater*:
  pgte a (*pmin a b*)
  by (*simp add*: *pgte.rep-eq pmin.rep-eq*)

lemma *pmin-is*:
  assumes *pgte a b*
  shows *pmin a b* = *b*
  by (*metis Rep-prat-inject assms min-absorb2 pgte.rep-eq pmin.rep-eq*)

lemma *pmax-comm*:
  pmax a b = *pmax b a*
  by (*metis Rep-prat-inverse max.commute pmax.rep-eq*)

lemma *pmax-smaller*:
  pgte (*pmax a b*) *a*
  by (*simp add*: *pgte.rep-eq pmax.rep-eq*)

lemma *pmax-is*:
  assumes *pgte a b*
  shows *pmax a b* = *a*
  by (*metis Rep-prat-inject assms max.absorb-iff1 pgte.rep-eq pmax.rep-eq*)

lemma *pmax-is-smaller*:
  assumes *pgte x a*
    and *pgte x b*
   shows *pgte x* (*pmax a b*)
proof (*cases pgte a b*)
  case *True*
  then show *?thesis*
   by (*simp add*: *assms*(*1*) *pmax-is*)
next
  case *False*
  then show *?thesis*
   using *assms*(*2*) *pgte.rep-eq pmax.rep-eq* **by** *auto*
qed

lemma *half-between-0-1*:
  pgt pwrite half
  by (*simp add*: *half.rep-eq pgt.rep-eq pwrite.rep-eq*)

lemma *pgt-implies-pgte*:
  assumes *pgt a b*
  shows *pgte a b*
  by (*meson assms less-imp-le pgt.rep-eq pgte.rep-eq*)

lemma *half-plus-half*:

padd half half = *pwrite*
by (*metis Rep-prat-inject divide-less-eq-numeral1*(*1*) *dual-order.irrefl half.rep-eq less-divide-eq-numeral1*(*1*) *linorder-neqE-linordered-idom mult.right-neutral one-add-one padd.rep-eq pwrite.rep-eq ring-class.ring-distribs*(*1*))

lemma *padd-comm*:
padd a b = *padd b a*
by (*metis Rep-prat-inject add.commute padd.rep-eq*)

lemma *padd-asso*:
padd (*padd a b*) *c* = *padd a* (*padd b c*)
by (*metis Rep-prat-inverse group-cancel.add1 padd.rep-eq*)

lemma *pgte-antisym*:
assumes *pgte a b*
and *pgte b a*
shows *a* = *b*
by (*metis Rep-prat-inverse assms*(*1*) *assms*(*2*) *leD le-less pgte.rep-eq*)

lemma *sum-larger*:
pgt (*padd a b*) *a*
using *Rep-prat padd.rep-eq pgt.rep-eq* **by** *auto*

lemma *greater-sum-both*:
assumes *pgte a* (*padd b c*)
shows $\exists$ *a1 a2*. *a* = *padd a1 a2* $\wedge$ *pgte a1 b* $\wedge$ *pgte a2 c*
proof −
obtain *aa bb cc* **where** *aa* = *Rep-prat a bb* = *Rep-prat b cc* = *Rep-prat c*
by *simp*
then have $aa \geq bb + cc$
using *assms padd.rep-eq pgte.rep-eq* **by** *auto*
then obtain *x* **where** $aa = bb + x$ $x \geq cc$
by (*metis add.commute add-le-cancel-left diff-add-cancel*)
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *Abs-prat-inverse Rep-prat Rep-prat-inverse* ‹*aa* = *Rep-prat a*› ‹*bb* = *Rep-prat b*› ‹*cc* = *Rep-prat c*› *dual-order.trans eq-onp-same-args le-less mem-Collect-eq min-absorb2 min-def order-refl padd.abs-eq pgte.rep-eq*)
qed

lemma *padd-cancellative*:
assumes *a* = *padd x b*
and *a* = *padd y b*
shows *x* = *y*
by (*metis Rep-prat-inject add-le-cancel-right assms*(*1*) *assms*(*2*) *leD less-eq-rat-def padd.rep-eq*)

lemma *not-pgte-charact*:

$\neg$ *pgte a b* $\longleftrightarrow$ *pgt b a*
by (*meson not-less pgt.rep-eq pgte.rep-eq*)

lemma *pgte-pgt*:
assumes *pgt a b*
and *pgte c d*
shows *pgt* (*padd a c*) (*padd b d*)
using *assms*(*1*) *assms*(*2*) *padd.rep-eq pgt.rep-eq pgte.rep-eq* **by** *auto*

lemma *pmult-distr*:
pmult a (*padd b c*) = *padd* (*pmult a b*) (*pmult a c*)
by (*metis Rep-prat-inject distrib-left padd.rep-eq pmult.rep-eq*)

lemma *pmult-comm*:
pmult a b = *pmult b a*
by (*metis Rep-prat-inject mult.commute pmult.rep-eq*)

lemma *pmult-special*:
pmult pwrite x = *x*
by (*metis Rep-prat-inverse comm-monoid-mult-class.mult-1 pmult.rep-eq pwrite.rep-eq*)

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

lemma *pinv-double-half*:
pmult half (*pinv p*) = *pinv* (*padd p p*)
proof –
have (*Fract 1 2*) ∗ ((*Fract 1 1*) / (*Rep-prat p*)) = (*Fract 1 1*) / (*Rep-prat p* + *Rep-prat p*)
by (*metis* (*no-types*, *lifting*) *One-rat-def comm-monoid-mult-class.mult-1 divide-rat mult-2 mult-rat rat-number-expand*(*3*) *times-divide-times-eq*)
then show *?thesis*
by (*metis Rep-prat-inject half.rep-eq mult-2 mult-numeral-1-right numeral-One padd.rep-eq pdiv.rep-eq pinv-def pmult.rep-eq pwrite.rep-eq times-divide-times-eq*)
qed

lemma *pinv-inverts*:
assumes *pgte a b*
shows *pgte* (*pinv b*) (*pinv a*)
proof –
have *Rep-prat a* $\geq$ *Rep-prat b*
using *assms*(*1*) *pgte.rep-eq* **by** *auto*
then have (*Fract 1 1*) / *Rep-prat b* $\geq$ (*Fract 1 1*) / *Rep-prat a*
by (*metis One-rat-def Rep-prat frac-le le-numeral-extra*(*4*) *mem-Collect-eq zero-le-one*)
then show *?thesis*
by (*simp add*: *One-rat-def pdiv.rep-eq pgte.rep-eq pinv-def pwrite.rep-eq*)

qed

lemma *pinv-pmult-ok*:
pmult p (*pinv p*) = *pwrite*
proof −
obtain *r* **where** *r* = *Rep-prat p* **by** *simp*
then have *r* ∗ ((*Fract 1 1*) / *r*) = *Fract 1 1*
by (*metis Rep-prat less-numeral-extra*(*3*) *mem-Collect-eq nonzero-mult-div-cancel-left times-divide-eq-right*)
then show *?thesis*
by (*metis One-rat-def Rep-prat-inject* ‹*r* = *Rep-prat p*› *pdiv.rep-eq pinv-def pmult.rep-eq pwrite.rep-eq*)
qed

lemma *pinv-pwrite*:
pinv pwrite = *pwrite*
by (*metis Rep-prat-inverse div-by-1 pdiv.rep-eq pinv-def pwrite.rep-eq*)

lemma *pmin-pmax*:
assumes *pgte x* (*pmin a b*)
shows *x* = *pmin* (*pmax x a*) (*pmax x b*)
proof (*cases pgte x a*)
case *True*
then show *?thesis*
by (*metis pmax-is pmax-smaller pmin-comm pmin-is*)
next
case *False*
then show *?thesis*
by (*metis assms not-pgte-charact pgt-implies-pgte pmax-is pmax-smaller pmin-comm pmin-is*)
qed

lemma *pmin-sum*:
padd (*pmin a b*) *c* = *pmin* (*padd a c*) (*padd b c*)
by (*metis not-pgte-charact pgt-implies-pgte pgte-pgt pmin-comm pmin-is*)

lemma *pmin-sum-larger*:
pgte (*pmin* (*padd a1 b1*) (*padd a2 b2*)) (*padd* (*pmin a1 a2*) (*pmin b1 b2*))
proof (*cases pgte* (*padd a1 b1*) (*padd a2 b2*))
case *True*
then have *pmin* (*padd a1 b1*) (*padd a2 b2*) = *padd a2 b2*
by (*simp add*: *pmin-is*)
moreover have *pgte a2* (*pmin a1 a2*) ∧ *pgte b2* (*pmin b1 b2*)
by (*metis pmin-comm pmin-greater*)
ultimately show *?thesis*
by (*simp add*: *padd.rep-eq pgte.rep-eq*)

next
  case *False*
  then have *pmin* (*padd a1 b1*) (*padd a2 b2*) = *padd a1 b1*
    by (*metis not-pgte-charact pgt-implies-pgte pmin-comm pmin-is*)
  moreover have *pgte a1* (*pmin a1 a2*) ∧ *pgte b1* (*pmin b1 b2*)
    by (*metis pmin-greater*)
  ultimately show *?thesis*
    by (*simp add*: *padd.rep-eq pgte.rep-eq*)
qed

end

1.3 Permission Heaps

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

theory *FractionalHeap*
  imports *Main PosRat PartialMap*
begin

type-synonym ($'l$, $'v$) *fract-heap* = $'l \rightharpoonup$ *prat* × $'v$

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

definition *compatible-fractions* :: ($'l$, $'v$) *fract-heap* ⇒ ($'l$, $'v$) *fract-heap* ⇒ *bool*
where
  compatible-fractions h h' ⟷
  (∀ *l p p'*. *h l* = *Some p* ∧ *h' l* = *Some p'* ⟶ *pgte pwrite* (*padd* (*fst p*) (*fst p'*)))

definition *same-values* :: ($'l$, $'v$) *fract-heap* ⇒ ($'l$, $'v$) *fract-heap* ⇒ *bool* **where**
  same-values h h' ⟷ (∀ *l p p'*. *h l* = *Some p* ∧ *h' l* = *Some p'* ⟶ *snd p* = *snd p'*)

fun *fadd-options* :: (*prat* × $'v$) *option* ⇒ (*prat* × $'v$) *option* ⇒ (*prat* × $'v$) *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 (*cases x*)
  case *None*
  then show *?thesis*
    by (*metis assms fadd-options.elims option.simps*(*3*))
next

case (*Some xx*)
then have $x = Some\ xx$ **by** *simp*
then show *?thesis*
apply (*cases a*)
apply (*cases b*)
apply *simp*
apply (*metis assms fadd-options.simps*(*1*) *fadd-options.simps*(*3*) *fst-conv not-pgte-charact option.sel padd-comm pgt-implies-pgte sum-larger*)
apply (*cases b*)
apply (*metis assms fadd-options.simps*(*1*) *fadd-options.simps*(*3*) *fst-conv not-pgte-charact option.sel padd-comm pgt-implies-pgte sum-larger*)

proof −
fix *aa bb* **assume** $a = Some\ aa$ $b = Some\ bb$
then have $snd\ aa = snd\ bb$
using *Some assms* **by** *auto*
moreover have $fst\ aa = fst\ bb$
using *padd-cancellative*[*of padd* (*fst aa*) (*fst xx*) *fst bb fst xx fst aa*]
Some ‹$a = Some\ aa$› ‹$b = Some\ bb$› *assms fadd-options.simps*(*3*) *fst-conv option.inject*
by *auto*
ultimately show $a = b$
by (*simp add*: ‹$a = Some\ aa$› ‹$b = Some\ bb$› *prod-eq-iff*)
qed
qed

definition *compatible-fract-heaps* :: ($'l$, $'v$) *fract-heap* ⇒ ($'l$, $'v$) *fract-heap* ⇒ *bool*
where
compatible-fract-heaps h h$'$ ⟷ *compatible-fractions h h*$'$ ∧ *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*$'$
by (*simp add*: *assms*(*1*) *assms*(*2*) *compatible-fract-heaps-def compatible-fractions-def same-values-def*)

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 $snd\ p = snd\ p'$
apply (*meson assms*(*1*) *assms*(*2*) *compatible-fract-heaps-def compatible-fractions-def*)
by (*meson assms*(*1*) *assms*(*2*) *compatible-fract-heaps-def same-values-def*)

lemma *compatible-fract-heaps-comm*:
assumes *compatible-fract-heaps h h*$'$

shows *compatible-fract-heaps h′ h*
proof (*rule compatible-fract-heapsI*)
show $\bigwedge l\ p\ p'.\ h'\ l = Some\ p \wedge h\ l = Some\ p' \Longrightarrow pgte\ pwrite\ (padd\ (fst\ p)\ (fst\ p'))$
by (*metis assms compatible-fract-heapsE*(*1*) *padd-comm*)
show $\bigwedge l\ p\ p'.\ h'\ l = Some\ p \wedge h\ l = Some\ p' \Longrightarrow snd\ p = snd\ p'$
using *assms compatible-fract-heapsE*(*2*) **by** *fastforce*
qed

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

definition *add-fh* :: ($'l$, $'v$) *fract-heap* ⇒ ($'l$, $'v$) *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 $\bigwedge l\ p.\ h\ l = Some\ p \Longrightarrow fst\ p = pwrite$
shows *full-ownership h*
by (*simp add*: *assms full-ownership-def*)

fun *apply-opt* **where**
apply-opt f None = *None*
| *apply-opt f* (*Some x*) = *Some* (*f x*)

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

definition *normalize* :: ($'l$, $'v$) *fract-heap* ⇒ ($'l \rightharpoonup 'v$) **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*)
apply (*metis FractionalHeap.normalize-def apply-opt.elims option.distinct*(*1*))
proof
show *?B* ⟹ *?A*
by (*metis FractionalHeap.normalize-def apply-opt.simps*(*2*) *snd-eqD*)
assume *?A* **then have** *h l* ≠ *None*
by (*metis FractionalHeap.normalize-def apply-opt.simps*(*1*) *option.distinct*(*1*))
then obtain *p* **where** *h l* = *Some p*
by *blast*
then show *?B*
by (*metis FractionalHeap.normalize-def* ‹*FractionalHeap.normalize h l* = *Some v*› ‹*h l* ≠ *None*› *apply-opt.elims option.inject prod.exhaust-sel*)
qed

```
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 −
  have r: ⋀h1 h2. compatible-fract-heaps h1 h2 ⟹ dom h1 ∩ fpdom h2 = {}
  proof −
    fix h1 h2 assume asm0: compatible-fract-heaps h1 h2
    show dom h1 ∩ fpdom h2 = {}
    proof
      show dom h1 ∩ fpdom h2 ⊆ {}
      proof
        fix x assume x ∈ dom h1 ∩ fpdom h2
        then have x ∈ dom h1 ∧ x ∈ fpdom h2 by auto
        then have h1 x ≠ None ∧ h2 x ≠ None
          using domIff fpdom-def[of h2] mem-Collect-eq option.discI
          by auto
        then obtain a b where h1 x = Some a h2 x = Some b by auto
        then have fst b = pwrite ∧ pgte pwrite (padd (fst a) (fst b))
            using ‹x ∈ dom h1 ∧ x ∈ fpdom h2› asm0 compatible-fract-heapsE(1)
fpdom-def[of h2] fst-conv mem-Collect-eq option.sel
          by fastforce
        then show x ∈ {}
          by (metis not-pgte-charact padd-comm sum-larger)
      qed
    qed (simp)
  qed
  then show dom h1 ∩ fpdom h2 = {}
    using assms by blast
  show dom h2 ∩ fpdom h1 = {}
    by (simp add: assms compatible-fract-heaps-comm r)
qed

lemma compatible-dom-sum:
  assumes compatible-fract-heaps h1 h2
  shows dom (add-fh h1 h2) = dom h1 ∪ dom h2 (is ?A = ?B)
proof
  show ?B ⊆ ?A
  proof
    fix x assume x ∈ ?B
    show x ∈ ?A
    proof (cases x ∈ dom h1)
      case True
     then show ?thesis using add-fh-def[of h1 h2] domI domIff fadd-options.elims
        by metis
    next
```

case *False*
then have $x \in$ *dom h2*
using ‹$x \in$ *dom h1* $\cup$ *dom h2*› **by** *auto*
then show *?thesis* **using** *add-fh-def*[*of h1 h2*] *domI domIff fadd-options.elims*
by *metis*
qed
qed
show *?A* $\subseteq$ *?B*
using *UnI1*[*of - dom h1 dom h2*] *UnI2*[*of - dom h1 dom h2*] *add-fh-def*[*of h1 h2*] *domIff fadd-options.simps*(*1*) *subset-iff*[*of ?A ?B*]
dom-map-add map-add-None
by *metis*
qed

Addition of permission heaps is associative.

lemma *add-fh-asso*:
add-fh (*add-fh a b*) *c* = *add-fh a* (*add-fh b c*)
proof (*rule ext*)
fix *x*
show *add-fh* (*add-fh a b*) *c x* = *add-fh a* (*add-fh b c*) *x*
proof (*cases a x*)
case *None*
then show *?thesis*
by (*simp add*: *add-fh-def*)
next
case (*Some aa*)
then have *a x* = *Some aa* **by** *simp*
then show *?thesis*
proof (*cases b x*)
case *None*
then show *?thesis*
by (*simp add*: *Some add-fh-def*)
next
case (*Some bb*)
then have *b x* = *Some bb* **by** *simp*
then show *?thesis*
proof (*cases c x*)
case *None*
then show *?thesis*
by (*simp add*: *Some* ‹*a x* = *Some aa*› *add-fh-def*)
next
case (*Some cc*)
then have *add-fh* (*add-fh a b*) *c x* = *Some* (*padd* (*padd* (*fst aa*) (*fst bb*)) (*fst cc*), *snd aa*)
by (*simp add*: ‹*a x* = *Some aa*› ‹*b x* = *Some bb*› *add-fh-def*)
moreover have *add-fh a* (*add-fh b c*) *x* = *Some* (*padd* (*fst aa*) (*padd* (*fst bb*) (*fst cc*)), *snd aa*)
by (*simp add*: *Some* ‹*a x* = *Some aa*› ‹*b x* = *Some bb*› *add-fh-def*)
ultimately show *?thesis*

```
      by (simp add: padd-asso)
    qed
   qed
  qed
qed

lemma add-fh-update:
  assumes b x = None
  shows add-fh (a(x ↦ p)) b = (add-fh a b)(x ↦ p)
proof (rule ext)
  fix l show add-fh (a(x ↦ p)) b l = ((add-fh a b)(x ↦ p)) l
    apply (cases l = x)
    apply (simp add: add-fh-def assms)
    by (simp add: add-fh-def)
qed

end
```

1.4 Extended Heaps

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

```
theory StateModel
  imports FractionalHeap HOL−Library.Multiset
begin

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

We store the initial value with the unique guard

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


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

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

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

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

The shared guards permissions sum to at most 1

definition *compatible* :: ('*i*, '*a*) *heap* ⇒ ('*i*, '*a*) *heap* ⇒ *bool* (**infixl** ‹##› *60*)
where
$h \ \#\# \ h' \longleftrightarrow$ *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'*
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *compatible-def* **by** *blast*

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 ⋀*k*. *h k* = *None* ∨ *h' k* = *None*
shows *add-gu h h'* = *add-gu h' h*
proof (*rule ext*)
fix *k* **show** *add-gu h h' k* = *add-gu h' h k*
by (*metis add-gu-def add-gu-single.simps*(*1*) *add-gu-single.simps*(*2*) *assms not-None-eq*)
qed

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

Addition of shared guard states is cancellative.

lemma *add-gs-cancellative*:
assumes *add-gs a x* = *add-gs b x*
shows *a* = *b*
apply (*cases x*)
apply (*metis add-gs.elims assms not-None-eq*)
apply (*cases a*)
apply (*cases b*)
apply *simp*
apply (*metis add-gs.simps*(*1*) *add-gs.simps*(*3*) *assms fst-conv not-pgte-charact*

option.sel padd-comm pgt-implies-pgte sum-larger)
  apply (*cases b*)
   apply (*metis add-gs.simps*(*1*) *add-gs.simps*(*3*) *assms fst-conv not-pgte-charact*
option.sel padd-comm pgt-implies-pgte sum-larger)
proof −
  fix *xx aa bb* **assume** *x* = *Some xx a* = *Some aa b* = *Some bb*
  then have *fst aa* = *fst bb*
    using *assms padd-cancellative*[*of padd* (*fst aa*) (*fst xx*)]
     Pair-inject add-gs.simps(*3*) *option.inject* **by** *auto*
  moreover have *snd aa* = *snd bb*
    using *add-left-cancel*[*of snd xx snd aa snd bb*]
    using ‹*a* = *Some aa*› ‹*b* = *Some bb*› ‹*x* = *Some xx*› *assms* **by** *auto*
  ultimately show *a* = *b*
    by (*simp add*: ‹*a* = *Some aa*› ‹*b* = *Some bb*› *prod-eq-iff*)
qed

Addition of shared guard states is commutative.

lemma *add-gs-comm*:
  add-gs a b = *add-gs b a*
proof (*cases a*)
  case *None*
  then show *?thesis*
    by (*metis add-gs.elims add-gs.simps*(*1*) *add-gs.simps*(*2*))
next
  case (*Some aa*)
  then have *a* = *Some aa* **by** *simp*
  then show *?thesis*
  proof (*cases b*)
    case *None*
    then show *?thesis*
      using *Some* **by** *force*
  next
    case (*Some bb*)
    moreover have *padd* (*fst aa*) (*fst bb*) = *padd* (*fst bb*) (*fst aa*) ∧ *snd aa* + *snd*
bb = *snd bb* + *snd aa*
      using *padd-comm* **by** *force*
    ultimately show *?thesis*
      using ‹*a* = *Some aa*› **by** *force*
  qed
qed

lemma *compatible-fheaps-comm*:
  assumes *compatible-fract-heaps a b*
  shows *add-fh a b* = *add-fh b a*
proof (*rule ext*)
  fix *x* **show** *add-fh a b x* = *add-fh b a x*
  proof (*cases a x*)
    case *None*
    then show *?thesis*

```
      by (metis add-fh-def add-fh-def fadd-options.simps(1) fadd-options.simps(2)
option.exhaust-sel)
  next
    case (Some aa)
    then have a x = Some aa by simp
    then show ?thesis
    proof (cases b x)
      case None
      then show ?thesis
        by (simp add: Some add-fh-def)
    next
      case (Some bb)
      then show ?thesis
        using ‹a x = Some aa› add-fh-def[of a b] add-fh-def[of b a] assms compati-
ble-fract-heapsE(2) fadd-options.simps(3) padd-comm
        by (metis (full-types))
    qed
  qed
qed
```

The following function defines addition between two extended heaps.

```
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)
   apply (metis assms eq-fst-iff get-fh.simps option.inject option.simps(3) plus.simps(3))
  apply (metis assms fst-eqD get-gs.simps option.distinct(1) option.inject plus.simps(3)
snd-conv)
 by (metis assms get-gu.elims option.distinct(1) option.sel plus.simps(3) snd-conv)

lemma compatible-eq:
  Some a ⊕ Some b = None ⟷ ¬ a ## b
  by simp

lemma compatible-comm:
  a ## b ⟷ b ## a
proof –
  have ⋀a b. a ## b ⟹ b ## a
  proof –
    fix a b assume asm0: a ## b
    show b ## a
```

proof (*rule compatibleI*)
show *compatible-fract-heaps* (*get-fh b*) (*get-fh a*)
using *asm0 compatible-def compatible-fract-heaps-comm* **by** *blast*
show $\bigwedge k.\ \textit{get-gu}\ b\ k = \textit{None} \vee \textit{get-gu}\ a\ k = \textit{None}$
by (*meson asm0 compatible-def*)
show $\bigwedge p\ p'.\ \textit{get-gs}\ b = \textit{Some}\ p \wedge \textit{get-gs}\ a = \textit{Some}\ p' \Longrightarrow \textit{pgte pwrite}\ (\textit{padd}\ (\textit{fst}\ p)\ (\textit{fst}\ p'))$
by (*metis asm0 compatible-def padd-comm*)
qed
qed
then show *?thesis*
by *blast*
qed

lemma *heap-ext*:
assumes $\textit{get-fh}\ a = \textit{get-fh}\ b$
and $\textit{get-gs}\ a = \textit{get-gs}\ b$
and $\textit{get-gu}\ a = \textit{get-gu}\ b$
shows $a = b$
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *get-fh.simps get-gs.simps get-gu.elims prod.expand*)

Addition of two extended heaps is commutative.

lemma *plus-comm*:
$a \oplus b = b \oplus a$
proof −
have *r*: $\bigwedge x\ a\ b.\ \textit{Some}\ x = \textit{Some}\ a \oplus \textit{Some}\ b \Longrightarrow \textit{Some}\ x = \textit{Some}\ b \oplus \textit{Some}\ a$
proof −
fix *x a b* **assume** *asm0*: $\textit{Some}\ x = \textit{Some}\ a \oplus \textit{Some}\ b$
then obtain *y* **where** $\textit{Some}\ y = \textit{Some}\ b \oplus \textit{Some}\ a$
by (*metis compatible-comm plus.simps*(*3*))
have $x = y$
proof (*rule heap-ext*)
show $\textit{get-fh}\ x = \textit{get-fh}\ y$
by (*metis* ‹$\textit{Some}\ y = \textit{Some}\ b \oplus \textit{Some}\ a$› *asm0 compatible-def compatible-eq compatible-fheaps-comm plus-extract*(*1*))
show $\textit{get-gs}\ x = \textit{get-gs}\ y$
by (*metis* ‹$\textit{Some}\ y = \textit{Some}\ b \oplus \textit{Some}\ a$› *add-gs-comm asm0 plus-extract*(*2*))
show $\textit{get-gu}\ x = \textit{get-gu}\ y$ **using** *comp-add-gu-comm*[*of get-gu x get-gu y*]
by (*metis* ‹$\textit{Some}\ y = \textit{Some}\ b \oplus \textit{Some}\ a$› *asm0 comp-add-gu-comm compatible-def compatible-eq plus-extract*(*3*))
qed
then show $\textit{Some}\ x = \textit{Some}\ b \oplus \textit{Some}\ a$
by (*simp add*: ‹$\textit{Some}\ y = \textit{Some}\ b \oplus \textit{Some}\ a$›)
qed
then show *?thesis*
proof (*cases* $a \oplus b$)
case *None*
then show *?thesis*

by (*metis* (*no-types*, *opaque-lifting*) *compatible-comm compatible-eq plus.elims*)
next
case (*Some ab*)
then have $a = Some\ (the\ a) \wedge b = Some\ (the\ b)$
by (*metis option.collapse option.distinct*(*1*) *plus.simps*(*1*) *plus.simps*(*2*))
then show *?thesis*
by (*metis* ‹$\bigwedge x\ b\ a.\ Some\ x = Some\ a \oplus Some\ b \Longrightarrow Some\ x = Some\ b \oplus Some\ a$› *plus.elims*)
qed
qed

lemma *asso2*:
assumes $Some\ a \oplus Some\ b = Some\ ab$
and $\neg\ b\ \#\#\ c$
shows $\neg\ ab\ \#\#\ c$
proof (*cases compatible-fract-heaps* (*get-fh b*) (*get-fh c*))
case *True*
then have *r*: $(\exists k.\ get\text{-}gu\ b\ k \neq None \wedge get\text{-}gu\ c\ k \neq None)$
$\vee\ (\exists p\ p'.\ get\text{-}gs\ b = Some\ p \wedge get\text{-}gs\ c = Some\ p' \wedge pgt\ (padd\ (fst\ p)\ (fst\ p'))\ pwrite)$
by (*metis assms*(*2*) *compatible-def not-pgte-charact*)
then show *?thesis*
proof (*cases* $\exists k.\ get\text{-}gu\ b\ k \neq None \wedge get\text{-}gu\ c\ k \neq None$)
case *True*
then obtain *k* **where** $get\text{-}gu\ b\ k \neq None \wedge get\text{-}gu\ c\ k \neq None$
by *auto*
then have $get\text{-}gu\ ab\ k \neq None$
using *add-gu-def*[*of get-gu a get-gu b*] *add-gu-single.simps*(*1*) *assms*(*1*) *compatible-def compatible-eq option.distinct*(*1*) *plus-extract*(*3*)
by *metis*
then show *?thesis*
by (*meson* ‹$get\text{-}gu\ b\ k \neq None \wedge get\text{-}gu\ c\ k \neq None$› *compatible-def*)
next
case *False*
then obtain $p\ p'$ **where** $get\text{-}gs\ b = Some\ p \wedge get\text{-}gs\ c = Some\ p' \wedge pgt\ (padd\ (fst\ p)\ (fst\ p'))\ pwrite$
using *r* **by** *blast*
moreover have $get\text{-}gs\ ab = add\text{-}gs\ (get\text{-}gs\ a)\ (Some\ p)$
by (*metis assms*(*1*) *calculation plus-extract*(*2*))
then show *?thesis*
proof (*cases get-gs a*)
case *None*
then show *?thesis*
by (*metis* ‹$get\text{-}gs\ ab = add\text{-}gs\ (get\text{-}gs\ a)\ (Some\ p)$› *add-gs.simps*(*1*) *calculation compatible-def not-pgte-charact*)
next
case (*Some pa*)
then have $get\text{-}gs\ ab = Some\ (padd\ (fst\ pa)\ (fst\ p),\ snd\ pa + snd\ p)$
using ‹$get\text{-}gs\ ab = add\text{-}gs\ (get\text{-}gs\ a)\ (Some\ p)$› **by** *auto*

then have *pgte* (*padd* (*fst pa*) (*fst p*)) (*fst p*)
using *padd-comm pgt-implies-pgte sum-larger* **by** *presburger*
then have *pgt* (*padd* (*padd* (*fst pa*) (*fst p*)) (*fst p'*)) *pwrite*
using *calculation padd.rep-eq pgt.rep-eq pgte.rep-eq* **by** *auto*
then show *?thesis*
by (*metis* ‹*get-gs ab* = *Some* (*padd* (*fst pa*) (*fst p*), *snd pa* + *snd p*)› *calculation compatible-def fst-conv not-pgte-charact*)
qed
qed
next
case *False*
then show *?thesis*
proof (*cases compatible-fractions* (*get-fh b*) (*get-fh c*))
case *True*
then have ¬ *same-values* (*get-fh b*) (*get-fh c*)
using *False compatible-fract-heaps-def* **by** *blast*
then obtain *l pb pc* **where** *get-fh b l* = *Some pb get-fh c l* = *Some pc snd pc* ≠ *snd pb*
using *same-values-def* **by** *fastforce*
then obtain *pab* **where** *get-fh ab l* = *Some pab snd pab* = *snd pb*
apply (*cases get-fh a l*)
apply (*metis* (*no-types*, *lifting*) *add-fh-def assms*(*1*) *fadd-options.simps*(*2*) *plus-comm plus-extract*(*1*))
using *add-fh-def*[*of get-fh b get-fh a l*] *assms*(*1*) *fadd-options.simps*(*3*) *plus-comm plus-extract*(*1*) *snd-conv*
by *metis*
then show *?thesis*
by (*metis* ‹*get-fh c l* = *Some pc*› ‹*snd pc* ≠ *snd pb*› *compatible-def compatible-fract-heapsE*(*2*))
next
case *False*
then obtain *pb pc l* **where** *get-fh b l* = *Some pb get-fh c l* = *Some pc pgt* (*padd* (*fst pb*) (*fst pc*)) *pwrite*
using *compatible-fractions-def not-pgte-charact* **by** *blast*

then show *?thesis*
proof (*cases get-fh a l*)
case *None*
then have *get-fh ab l* = *Some pb*
by (*metis* (*no-types*, *lifting*) ‹*get-fh b l* = *Some pb*› *add-fh-def assms*(*1*) *fadd-options.simps*(*1*) *plus-extract*(*1*))
then show *?thesis*
by (*meson* ‹*get-fh c l* = *Some pc*› ‹*pgt* (*padd* (*fst pb*) (*fst pc*)) *pwrite*› *compatible-def compatible-fract-heaps-def compatible-fractions-def not-pgte-charact*)
next
case (*Some pa*)
then obtain *pab* **where** *get-fh ab l* = *Some pab fst pab* = *padd* (*fst pa*) (*fst pb*)
by (*metis* (*mono-tags*, *opaque-lifting*) ‹*get-fh b l* = *Some pb*› *add-fh-def*

assms(*1*) *fadd-options.simps*(*3*) *fst-conv plus-extract*(*1*))
then have *pgte* (*fst pab*) (*fst pb*)
by (*metis padd-comm pgt-implies-pgte sum-larger*)
then have *pgt* (*padd* (*fst pab*) (*fst pc*)) *pwrite*
using ‹*pgt* (*padd* (*fst pb*) (*fst pc*)) *pwrite*› *padd.rep-eq pgt.rep-eq pgte.rep-eq* **by** *force*
then show *?thesis*
by (*meson* ‹*get-fh ab l* = *Some pab*› ‹*get-fh c l* = *Some pc*› *compatible-def compatible-fract-heapsE*(*1*) *not-pgte-charact*)
qed
qed
qed

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*)
using *add-fh-def*[*of get-fh a get-fh b*] *assms*(*1*) *assms*(*2*) *assms*(*3*) *compatible-def*[*of a b*] *compatible-eq*
compatible-fract-heapsE(*1*)[*of get-fh a get-fh b*] *compatible-fract-heapsE*(*2*)[*of get-fh a get-fh b*]
fadd-options.simps(*3*)[*of pa pb*] *option.distinct*(*1*) *plus-extract*(*1*)[*of x a b*]
by *metis*

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 (*cases Some ab* ⊕ *Some c*)
case *None*
then show *?thesis*
proof (*cases compatible-fract-heaps* (*get-fh ab*) (*get-fh c*))
case *True*
then have *r*: (∃ *k*. *get-gu ab k* ≠ *None* ∧ *get-gu c k* ≠ *None*) ∨ (∃ *p p'*. *get-gs ab* = *Some p* ∧ *get-gs c* = *Some p'*
∧ *pgt* (*padd* (*fst p*) (*fst p'*)) *pwrite*)
by (*metis None compatible-def compatible-eq not-pgte-charact*)
then show *?thesis*
proof (*cases* ∃ *k*. *get-gu ab k* ≠ *None* ∧ *get-gu c k* ≠ *None*)
case *True*
then obtain *k* **where** *get-gu ab k* ≠ *None* ∧ *get-gu c k* ≠ *None*
by *presburger*
then have *get-gu a k* ≠ *None* ∨ *get-gu b k* ≠ *None*
by (*metis* (*no-types*, *lifting*) *add-gu-def add-gu-single.simps*(*1*) *assms*(*1*) *plus-extract*(*3*))
then show *?thesis*
by (*metis* ‹*get-gu ab k* ≠ *None* ∧ *get-gu c k* ≠ *None*› *assms*(*2*) *asso2*

compatible-def compatible-eq option.discI plus-comm)
next
case *False*
then obtain *pab pc* **where** *get-gs ab* = *Some pab* ∧ *get-gs c* = *Some pc* ∧ *pgt* (*padd* (*fst pab*) (*fst pc*)) *pwrite*
using *r* **by** *blast*
then show *?thesis*
apply (*cases get-gs a*)
apply (*metis add-gs.simps*(*1*) *assms*(*1*) *assms*(*2*) *compatible-def compatible-eq not-pgte-charact option.discI plus-extract*(*2*))
apply (*cases get-gs b*)
apply (*metis add-gs.simps*(*1*) *add-gs.simps*(*2*) *assms*(*1*) *assms*(*2*) *compatible-def compatible-eq not-pgte-charact plus-extract*(*2*))
proof −
fix *pa pb* **assume** *asm*: *get-gs ab* = *Some pab* ∧ *get-gs c* = *Some pc* ∧ *pgt* (*padd* (*fst pab*) (*fst pc*)) *pwrite*
get-gs a = *Some pa get-gs b* = *Some pb*
then have *pab* = (*padd* (*fst pa*) (*fst pb*), *snd pa* + *snd pb*)
by (*metis add-gs.simps*(*3*) *assms*(*1*) *option.sel plus-extract*(*2*))
then show *Some ab* ⊕ *Some c* = *Some a* ⊕ *Some bc*
using *None* ‹*get-gs a* = *Some pa*› *asm*
‹*get-gs b* = *Some pb*› *add-gs.simps*(*3*) *assms*(*2*) *compatible-def*[*of a bc*]
compatible-eq fst-conv not-pgte-charact[*of pwrite padd* (*fst pab*) (*fst pc*)] *padd-asso plus-extract*(*2*)
by *metis*
qed
qed
next
case *False*
then show *?thesis*
proof (*cases compatible-fractions* (*get-fh ab*) (*get-fh c*))
case *True*
then have ¬*same-values* (*get-fh ab*) (*get-fh c*)
using *False compatible-fract-heaps-def*
by *blast*

then obtain *l pab pc* **where** *get-fh ab l* = *Some pab get-fh c l* = *Some pc snd pab* ≠ *snd pc*
using *same-values-def* **by** *blast*

then show *?thesis*
apply (*cases get-fh a l*)

apply (*metis* (*no-types*, *lifting*) *add-fh-def assms*(*1*) *assms*(*2*) *compatible-def compatible-eq compatible-fract-heapsE*(*2*) *fadd-options.simps*(*1*) *option.distinct*(*1*) *plus-extract*(*1*))

proof −
fix *pa* **assume** *get-fh ab l* = *Some pab get-fh c l* = *Some pc snd pab* ≠ *snd*

pc get-fh a l = Some pa
moreover have *same-values (get-fh a) (get-fh b)*
by (*metis assms(1) compatible-def compatible-fract-heaps-def option.discI plus.simps(3)*)
ultimately have *snd pa = snd pab*
apply (*cases get-fh b l*)
apply (*metis (no-types, lifting) add-fh-def assms(1) fadd-options.simps(2) option.inject plus-extract(1)*)
by (*metis (no-types, lifting) add-fh-def assms(1) fadd-options.simps(3) option.sel plus-extract(1) snd-eqD*)
then show *?thesis*
by (*metis (full-types) None ‹get-fh a l = Some pa› ‹get-fh c l = Some pc› ‹snd pab ≠ snd pc› assms(2) asso2 compatible-def compatible-eq compatible-fract-heapsE(2) plus-comm*)
qed
next
case *False*
then obtain *l pab pc* **where** *get-fh ab l = Some pab get-fh c l = Some pc pgt (padd (fst pab) (fst pc)) pwrite*
using *compatible-fractions-def not-pgte-charact* **by** *blast*
then show *?thesis*
proof (*cases get-fh a l*)
case *None*
then have *get-fh b l = Some pab*
by (*metis (no-types, lifting) ‹get-fh ab l = Some pab› add-fh-def assms(1) fadd-options.simps(1) plus-extract(1)*)
then show *?thesis*
by (*metis ‹get-fh c l = Some pc› ‹pgt (padd (fst pab) (fst pc)) pwrite› assms(2) compatible-def compatible-fract-heapsE(1) not-pgte-charact option.simps(3) plus.simps(3)*)
next
case (*Some pa*)
then have *get-fh a l = Some pa* **by** *simp*
then show *?thesis*
proof (*cases get-fh b l*)
case *None*
then have *pa = pab*
by (*metis (no-types, lifting) Some ‹get-fh ab l = Some pab› add-fh-def assms(1) fadd-options.simps(2) option.inject plus-extract(1)*)
then show *?thesis*
by (*metis Some ‹get-fh ab l = Some pab› ‹get-fh c l = Some pc› ‹pgt (padd (fst pab) (fst pc)) pwrite› assms(2) asso2 compatible-def compatible-fract-heapsE(1) not-pgte-charact padd-comm plus.simps(3) plus-comm*)
next
case (*Some pb*)
then have *fst pab = padd (fst pa) (fst pb)*
using *‹get-fh a l = Some pa› ‹get-fh ab l = Some pab› add-fh-def[of get-fh a get-fh b] assms(1) compatible-def compatible-eq compatible-fract-heapsE(2)[of get-fh a get-fh b] fadd-options.simps(3)*

fst-apfst option.discI option.sel plus-extract(*1*)[*of ab a b*] *prod.collapse snd-apfst*
by *force*
then have *pgt* (*padd* (*fst pa*) (*padd* (*fst pb*) (*fst pc*))) *pwrite*
using ‹*pgt* (*padd* (*fst pab*) (*fst pc*)) *pwrite*› *padd-asso* **by** *auto*
moreover obtain *pbc* **where** *get-fh bc l* = *Some pbc fst pbc* = *padd* (*fst pb*) (*fst pc*)
by (*metis* (*no-types*, *opaque-lifting*) *Some* ‹*get-fh c l* = *Some pc*› *add-fh-def assms*(*2*) *fadd-options.simps*(*3*) *fst-conv plus-extract*(*1*))
ultimately show *?thesis*
by (*metis None* ‹*get-fh a l* = *Some pa*› *compatible-def compatible-eq compatible-fract-heapsE*(*1*) *not-pgte-charact*)
qed
qed
qed
qed
next
case (*Some x*)
then have *Some ab* ⊕ *Some c* = *Some x* **by** *simp*
have *a* ## *bc*
proof (*rule compatibleI*)
show *compatible-fract-heaps* (*get-fh a*) (*get-fh bc*)
proof (*rule compatible-fract-heapsI*)
fix *l pa pbc* **assume** *asm0*: *get-fh a l* = *Some pa* ∧ *get-fh bc l* = *Some pbc*
have *pgte pwrite* (*padd* (*fst pa*) (*fst pbc*)) ∧ *snd pa* = *snd pbc*
proof (*cases get-fh c l*)
case *None*
then have *get-fh b l* = *Some pbc*
by (*metis* (*no-types*, *lifting*) *add-fh-def asm0 assms*(*2*) *fadd-options.elims option.discI plus-extract*(*1*))
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *asm0 assms*(*1*) *compatible-def compatible-eq compatible-fract-heapsE*(*1*) *compatible-fract-heapsE*(*2*) *option.discI*)
next
case (*Some pc*)
then have *get-fh c l* = *Some pc* **by** *simp*
then show *?thesis*
proof (*cases get-fh b l*)
case *None*
then have *get-fh ab l* = *Some pa*
by (*metis* (*no-types*, *lifting*) *add-fh-def asm0 assms*(*1*) *fadd-options.simps*(*2*) *plus-extract*(*1*))
moreover have *pbc* = *pc*
by (*metis* (*no-types*, *lifting*) *None Some add-fh-def asm0 assms*(*2*) *fadd-options.simps*(*2*) *option.inject plus-comm plus-extract*(*1*))
ultimately show *?thesis*
by (*metis* (*no-types*, *lifting*) *Some* ‹*Some ab* ⊕ *Some c* = *Some x*› *compatible-def compatible-eq compatible-fract-heapsE*(*1*) *compatible-fract-heapsE*(*2*) *option.discI*)
next

case (*Some pb*)
then obtain *pab* **where** *get-fh ab l = Some pab fst pab = padd (fst pa) (fst pb) snd pab = snd pa*
by (*metis* (*mono-tags, opaque-lifting*) *add-fh-def asm0 assms*(*1*) *fadd-options.simps*(*3*) *fst-conv plus-extract*(*1*) *snd-conv*)
then have *pgte pwrite* (*padd* (*padd* (*fst pa*) (*fst pb*)) (*fst pc*))
by (*metis* ‹*Some ab* ⊕ *Some c = Some x*› ‹*get-fh c l = Some pc*› *compatible-def compatible-eq compatible-fract-heapsE*(*1*) *option.distinct*(*1*))
then have *pgte pwrite* (*padd* (*fst pa*) (*fst pbc*))
by (*metis* (*no-types, lifting*) *Some* ‹*get-fh c l = Some pc*› *add-fh-def asm0 assms*(*2*) *fadd-options.simps*(*3*) *fst-conv option.sel padd-asso plus-extract*(*1*))
moreover have *snd pa = snd pb*
by (*metis Some asm0 assms*(*1*) *compatible-def compatible-fract-heapsE*(*2*) *option.simps*(*3*) *plus.simps*(*3*))
then have *snd pa = snd pbc*
by (*metis* (*no-types, opaque-lifting*) *Some* ‹*get-fh c l = Some pc*› *add-fh-def asm0 assms*(*2*) *fadd-options.simps*(*3*) *option.sel plus-extract*(*1*) *snd-conv*)
ultimately show *?thesis* **by** *blast*
qed
qed
then show *pgte pwrite* (*padd* (*fst pa*) (*fst pbc*))
by *auto*
show *snd pa = snd pbc*
by (*simp add*: ‹*pgte pwrite* (*padd* (*fst pa*) (*fst pbc*)) ∧ *snd pa = snd pbc*›)
qed

show ⋀*k. get-gu a k = None* ∨ *get-gu bc k = None*
proof −
fix *k* **show** *get-gu a k = None* ∨ *get-gu bc k = None*
proof (*cases get-gu a k*)
case (*Some aa*)
then have *get-gu b k = None* ∨ *get-gu c k = None*
by (*metis assms*(*2*) *compatible-def compatible-eq option.discI*)
then show *?thesis*
using *Some* ‹*Some ab* ⊕ *Some c = Some x*› *add-gu-def*[*of get-gu a get-gu b*]
add-gu-def[*of get-gu b get-gu c*] *add-gu-single.simps*(*1*) *add-gu-single.simps*(*2*) *assms*(*1*) *assms*(*2*) *compatible-def compatible-eq option.distinct*(*1*) *plus-extract*(*3*)
by *metis*
qed (*simp*)
qed
fix *pa pbc* **assume** *get-gs a = Some pa* ∧ *get-gs bc = Some pbc*
show *pgte pwrite* (*padd* (*fst pa*) (*fst pbc*))
proof (*cases get-gs b*)
case *None*
then show *?thesis* **by** (*metis Some* ‹*get-gs a = Some pa* ∧ *get-gs bc = Some pbc*› *add-gs.simps*(*1*) *add-gs.simps*(*2*) *assms*(*1*) *assms*(*2*) *compatible-def compatible-eq option.discI plus-extract*(*2*))

next
case (*Some pb*)
then have *get-gs b* = *Some pb* **by** *simp*
then show *?thesis*
proof (*cases get-gs c*)
case *None*
then show *?thesis*
by (*metis Some* ‹*get-gs a* = *Some pa* ∧ *get-gs bc* = *Some pbc*› *add-gs.simps*(*2*) *assms*(*1*) *assms*(*2*) *compatible-def compatible-eq option.distinct*(*1*) *plus-extract*(*2*))
next
case (*Some pc*)
then have *padd* (*fst pa*) (*fst pbc*) = *padd* (*fst pa*) (*padd* (*fst pb*) (*fst pc*))
by (*metis* (*no-types*, *lifting*) ‹*get-gs a* = *Some pa* ∧ *get-gs bc* = *Some pbc*› ‹*get-gs b* = *Some pb*› *add-gs.simps*(*3*) *assms*(*2*) *fst-conv option.sel plus-extract*(*2*))
also have ... = *padd* (*padd* (*fst pa*) (*fst pb*)) (*fst pc*)
using *padd-asso* **by** *force*
moreover obtain *pab* **where** *get-gs ab* = *Some pab*
by (*metis* ‹*get-gs a* = *Some pa* ∧ *get-gs bc* = *Some pbc*› ‹*get-gs b* = *Some pb*› *add-gs.simps*(*3*) *assms*(*1*) *plus-extract*(*2*))
then have *pgte pwrite* (*padd* (*fst pab*) (*fst pc*))
by (*metis Some* ‹*Some ab* ⊕ *Some c* = *Some x*› *compatible-def compatible-eq option.simps*(*3*))
ultimately show *?thesis*
by (*metis* (*no-types*, *lifting*) ‹*get-gs a* = *Some pa* ∧ *get-gs bc* = *Some pbc*› ‹*get-gs ab* = *Some pab*› ‹*get-gs b* = *Some pb*› *add-gs.simps*(*3*) *assms*(*1*) *fst-conv option.sel plus-extract*(*2*))
qed
qed

qed
then obtain *y* **where** *Some y* = *Some a* ⊕ *Some bc*
by *simp*
moreover have $x = y$
proof (*rule heap-ext*)
show *get-gu x* = *get-gu y*
proof (*rule ext*)
fix *k* **show** *get-gu x k* = *get-gu y k*
apply (*cases get-gu a k*)
using *Some add-gu-def*[*of get-gu a*] *add-gu-def*[*of get-gu b*] *add-gu-def*[*of get-gu ab*]
add-gu-single.simps(*1*) *assms*(*1*) *assms*(*2*) *calculation*
plus-extract(*3*)[*of ab a b*] *plus-extract*(*3*)[*of bc b c*] *plus-extract*(*3*)[*of y a bc*] *plus-extract*(*3*)[*of x ab c*]
apply *simp*
apply (*cases get-gu b k*)
using *Some add-gu-def*[*of get-gu a*] *add-gu-def*[*of get-gu b*] *add-gu-def*[*of get-gu ab*]
add-gu-single.simps(*1*) *assms*(*1*) *assms*(*2*) *calculation*
plus-extract(*3*)[*of ab a b*] *plus-extract*(*3*)[*of bc b c*] *plus-extract*(*3*)[*of y a*

bc] *plus-extract*(*3*)[*of x ab c*]
add-gu-single.simps(*1*) *add-gu-single.simps*(*2*) *assms*(*1*) *assms*(*2*) *calculation*
apply *simp*
by (*metis assms*(*1*) *compatible-def compatible-eq option.simps*(*3*))
qed
show *get-gs x* = *get-gs y*
apply (*cases get-gs a*)
apply (*metis* (*mono-tags*, *lifting*) *Some add-gs.simps*(*1*) *assms*(*1*) *assms*(*2*) *calculation plus-extract*(*2*))
apply (*cases get-gs b*)
apply (*metis* (*mono-tags*, *lifting*) *Some add-gs.simps*(*1*) *add-gs.simps*(*2*) *assms*(*1*) *assms*(*2*) *calculation plus-extract*(*2*))
apply (*cases get-gs c*)
apply (*metis Some add-gs.simps*(*1*) *assms*(*1*) *assms*(*2*) *calculation plus-comm plus-extract*(*2*))
proof −
fix *ga gb gc* **assume** *asm0*: *get-gs a* = *Some ga get-gs b* = *Some gb get-gs c* = *Some gc*
then obtain *gab gbc* **where** *r*: *get-gs ab* = *Some gab get-gs bc* = *gbc*
by (*metis add-gs.simps*(*3*) *assms*(*1*) *plus-extract*(*2*))
then have *get-gs x* = *Some* (*padd* (*padd* (*fst ga*) (*fst gb*)) (*fst gc*), (*snd ga* + *snd gb*) + *snd gc*)
by (*metis* (*no-types*, *lifting*) *Some add-gs.simps*(*3*) *asm0*(*1*) *asm0*(*2*) *asm0*(*3*) *assms*(*1*) *fst-conv plus-extract*(*2*) *snd-conv*)
moreover have *get-gs y* = *Some* (*padd* (*fst ga*) (*padd* (*fst gb*) (*fst gc*)), *snd ga* + (*snd gb* + *snd gc*))
by (*metis* (*mono-tags*, *opaque-lifting*) ‹*Some y* = *Some a* ⊕ *Some bc*› *add-gs.simps*(*3*) *asm0*(*1*) *asm0*(*2*) *asm0*(*3*) *assms*(*2*) *fst-conv plus-extract*(*2*) *prod.exhaust-sel snd-conv*)
ultimately show *get-gs x* = *get-gs y*
by (*simp add*: *padd-asso*)
qed
show *get-fh x* = *get-fh y*
by (*metis Some add-fh-asso assms*(*1*) *assms*(*2*) *calculation plus-extract*(*1*))
qed
ultimately show *?thesis* **using** *Some* **by** *presburger*
qed

lemma *simpler-asso*:
(*Some a* ⊕ *Some b*) ⊕ *Some c* = *Some a* ⊕ (*Some b* ⊕ *Some c*)
proof (*cases Some a* ⊕ *Some b*)
case *None*
then show *?thesis*
by (*metis* (*no-types*, *opaque-lifting*) *asso2 compatible-eq option.exhaust plus.simps*(*1*) *plus-comm*)
next
case (*Some ab*)
then have *ab*: *Some ab* = *Some a* ⊕ *Some b* **by** *simp*

then show *?thesis*
proof (*cases Some b* ⊕ *Some c*)
case *None*
then show *?thesis*
by (*metis Some asso2 compatible-eq plus.simps*(*2*))
next
case (*Some bc*)
then show *?thesis*
by (*metis ab asso1*)
qed
qed

Addition of two extended heaps is associative.

lemma *plus-asso*:
($a \oplus b) \oplus c = a \oplus (b \oplus c)$
proof (*cases a*)
case (*Some aa*)
then have *aa*: *a* = *Some aa* **by** *simp*
then show *?thesis*
proof (*cases b*)
case (*Some bb*)
then have *bb*: *b* = *Some bb* **by** *simp*
then show *?thesis*
proof (*cases c*)
case *None*
then show *?thesis*
by (*simp add*: *plus-comm*)
next
case (*Some cc*)
then show *?thesis*
using *aa bb simpler-asso* **by** *blast*
qed
qed (*simp*)
qed (*simp*)

We define the extension order between extended heaps.

definition *larger* :: ('*i*, '*a*) *heap* ⇒ ('*i*, '*a*) *heap* ⇒ *bool* (**infixl** ‹⪰› *55*) **where**
$a \succeq b \longleftrightarrow (\exists c.\ Some\ a = Some\ b \oplus Some\ c)$

The extension order between extended heaps is transitive.

lemma *larger-trans*:
assumes $a \succeq b$
and $b \succeq c$
shows $a \succeq c$
proof −
obtain *r1* **where** *Some a* = *Some b* ⊕ *Some r1*
using *assms*(*1*) *larger-def* **by** *blast*
moreover obtain *r2* **where** *Some b* = *Some c* ⊕ *Some r2*
using *assms*(*2*) *larger-def* **by** *blast*

moreover obtain *r* **where** *Some r* = *Some r1* ⊕ *Some r2*
by (*metis* (*no-types*, *opaque-lifting*) *calculation*(*1*) *calculation*(*2*) *not-Some-eq plus.simps*(*1*) *plus-asso plus-comm*)
ultimately show *?thesis*
by (*metis larger-def plus-comm simpler-asso*)
qed

lemma *comp-smaller*:
assumes *a* ## *b*
and *Some b* = *Some c* ⊕ *Some d*
shows *a* ## *c*
by (*metis assms*(*1*) *assms*(*2*) *option.distinct*(*1*) *plus.simps*(*1*) *plus.simps*(*3*) *plus-asso*)

lemma *full-sguard-sum-same*:
assumes *get-gs a* = *Some* (*pwrite*, *sargs*)
and *Some h* = *Some a* ⊕ *Some b*
shows *get-gs h* = *Some* (*pwrite*, *sargs*)
proof (*cases get-gs b*)
case *None*
then show *?thesis*
by (*metis add-gs.simps*(*2*) *assms*(*1*) *assms*(*2*) *fst-conv get-gs.elims option.sel option.simps*(*3*) *plus.simps*(*3*) *snd-eqD*)
next
case (*Some a*)
then show *?thesis*
by (*metis assms*(*1*) *assms*(*2*) *compatible-def compatible-eq fst-eqD not-pgte-charact option.simps*(*3*) *sum-larger*)
qed

lemma *full-uguard-sum-same*:
assumes *get-gu a k* = *Some uargs*
and *Some h* = *Some a* ⊕ *Some b*
shows *get-gu h k* = *Some uargs*
proof (*cases get-gu b k*)
case *None*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *add-gu-def add-gu-single.simps*(*2*) *assms*(*1*) *assms*(*2*) *plus-extract*(*3*))
next
case (*Some a*)
then show *?thesis*
by (*metis assms*(*1*) *assms*(*2*) *compatible-def compatible-eq option.simps*(*3*))
qed

lemma *smaller-more-compatible*:
assumes *a* ## *b*
and *a* ⪰ *c*
shows *c* ## *b*

by (*meson assms*(*1*) *assms*(*2*) *comp-smaller compatible-comm larger-def*)

lemma *equiv-sum-get-fh*:
assumes *get-fh a* = *get-fh a′*
and *get-fh b* = *get-fh b′*
and *Some x* = *Some a* ⊕ *Some b*
and *Some x′* = *Some a′* ⊕ *Some b′*
shows *get-fh x* = *get-fh x′*
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *fst-eqD get-fh.elims option.discI option.sel plus.simps*(*3*))

lemma *addition-cancellative*:
assumes *Some a* = *Some b* ⊕ *Some c*
and *Some a* = *Some b′* ⊕ *Some c*
shows *b* = *b′*
proof (*rule heap-ext*)
show *get-gu b* = *get-gu b′*
proof (*rule ext*)
fix *k* **show** *get-gu b k* = *get-gu b′ k*
apply (*cases get-gu a k*)
apply (*metis assms*(*1*) *assms*(*2*) *full-uguard-sum-same not-Some-eq*)
apply (*cases get-gu b k*)
using *add-gu-def*[*of get-gu b get-gu c*]
add-gu-single.simps(*1*)[*of get-gu c k*] *assms*(*1*) *assms*(*2*) *compatible-def*[*of b c*] *compatible-def*[*of b′ c*]
option.inject option.simps(*3*) *plus.elims plus-extract*(*3*)[*of a b c*]
apply *metis*
proof −
fix *ga gb* **assume** *get-gu a k* = *Some ga get-gu b k* = *Some gb*
then have *get-gu c k* = *None*
by (*metis assms*(*1*) *compatible-def compatible-eq option.simps*(*3*))
then show *get-gu b k* = *get-gu b′ k*
by (*metis* (*no-types*, *opaque-lifting*) *add-gu-def add-gu-single.simps*(*1*) *assms*(*1*) *assms*(*2*) *plus-comm plus-extract*(*3*))
qed
qed
show *get-gs b* = *get-gs b′*
by (*metis add-gs-cancellative assms*(*1*) *assms*(*2*) *plus-extract*(*2*))
show *get-fh b* = *get-fh b′*
proof (*rule ext*)
fix *l* **show** *get-fh b l* = *get-fh b′ l*
proof (*cases get-fh a l*)
case *None*
then have *get-fh b l* = *None*
by (*metis* (*no-types*, *lifting*) *add-fh-def assms*(*1*) *fadd-options.elims option.distinct*(*1*) *plus-extract*(*1*))
then show *?thesis*
by (*metis* (*no-types*, *opaque-lifting*) *None add-fh-def assms*(*2*) *fadd-options.elims option.distinct*(*1*) *plus-extract*(*1*))

next
case (*Some aa*)
then have *get-fh a l* = *Some aa* **by** *simp*
then show *?thesis*
proof (*cases get-fh c l*)
case *None*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *add-fh-def assms*(*1*) *assms*(*2*) *fadd-options.simps*(*1*) *plus-comm plus-extract*(*1*))
next
case (*Some cc*)
then have *get-fh c l* = *Some cc* **by** *simp*
then show *?thesis* **using** *fadd-options-cancellative*
by (*metis* (*no-types*, *opaque-lifting*) *add-fh-def assms*(*1*) *assms*(*2*) *plus-extract*(*1*))
qed
qed
qed
qed

lemma *addition-cancellative3*:
assumes *Some x* = *Some a* ⊕ *Some b* ⊕ *Some c*
and *Some x* = *Some a′* ⊕ *Some b* ⊕ *Some c*
shows *a* = *a′*
proof −
obtain *ab ab′* **where** *Some ab* = *Some a* ⊕ *Some b Some ab′* = *Some a′* ⊕ *Some b*
by (*metis assms*(*1*) *assms*(*2*) *not-Some-eq plus.simps*(*1*))
then have *ab* = *ab′*
by (*metis addition-cancellative assms*(*1*) *assms*(*2*))
then show *?thesis*
using ‹*Some ab* = *Some a* ⊕ *Some b*› ‹*Some ab′* = *Some a′* ⊕ *Some b*› *addition-cancellative* **by** *blast*
qed

lemma *larger3*:
assumes *Some x* = *Some a* ⊕ *Some b* ⊕ *Some c*
shows *x* ⪰ *b*
proof −
obtain *ab* **where** *Some ab* = *Some a* ⊕ *Some b*
by (*metis assms not-Some-eq plus.simps*(*1*))
then show *?thesis*
by (*metis* (*no-types*, *opaque-lifting*) *assms larger-def larger-trans plus-comm*)
qed

lemma *add-get-fh*:

assumes *Some x* = *Some a* ⊕ *Some b*
shows *get-fh x* = *add-fh* (*get-fh a*) (*get-fh b*)
by (*metis assms fst-conv get-fh.elims option.discI option.sel plus.simps*(*3*))

lemma *sum-gs-one-none*:
assumes *Some x* = *Some a* ⊕ *Some b*
and *get-gs b* = *None*
shows *get-gs x* = *get-gs a*
by (*metis add-gs.simps*(*1*) *assms*(*1*) *assms*(*2*) *plus-comm plus-extract*(*2*))

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*)
by (*metis add-gs.simps*(*3*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *fst-conv plus-extract*(*2*) *snd-conv*)

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

lemma *dom-normalize*:
dom h = *dom* (*normalize h*)
by (*meson FractionalHeap.normalize-eq*(*1*) *domIff subsetI subset-antisym*)

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 (*cases get-fh a l*)
case *None*
then show *?thesis*
by (*metis* (*no-types*, *opaque-lifting*) *add-fh-def add-get-fh assms*(*1*) *assms*(*2*) *fadd-options.simps*(*1*))
next
case (*Some aa*)
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *add-fh-def add-get-fh assms*(*1*) *assms*(*2*) *fadd-options.simps*(*2*))
qed

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*
by (*metis assms*(*1*) *assms*(*2*) *plus-comm sum-second-none-get-fh*)

lemma *dom-sum-two*:
  assumes *Some x* = *Some a* ⊕ *Some b*
  shows *dom* (*get-fh x*) = *dom* (*get-fh a*) ∪ *dom* (*get-fh b*)
  by (*metis add-get-fh assms compatible-def compatible-dom-sum compatible-eq option.distinct*(*1*))

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 −
  obtain *ab* **where** *Some ab* = *Some a* ⊕ *Some b*
    by (*metis assms not-Some-eq plus.simps*(*1*))
  then have *Some x* = *Some ab* ⊕ *Some c*
    using *assms* **by** *presburger*
  then have *dom* (*get-fh x*) = *dom* (*get-fh ab*) ∪ *dom* (*get-fh c*)
    by (*meson dom-sum-two*)
  then show *?thesis*
    by (*metis* ‹*Some ab* = *Some a* ⊕ *Some b*› *dom-sum-two*)
qed

lemma *addition-smaller-domain*:
  assumes *Some a* = *Some b* ⊕ *Some c*
  shows *dom* (*get-fh b*) ⊆ *dom* (*get-fh a*)
  by (*metis* (*no-types*, *opaque-lifting*) *Un-subset-iff assms dom-sum-two order-refl*)

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*)
   using *assms*(*1*) *assms*(*2*) *not-Some-eq plus-extract-point-fh*[*of x a - l* (*π*, *v*)] *snd-eqD sum-second-none-get-fh*[*of x a*]
  by *metis*

lemma *compatible-last-two*:
  assumes *Some x* = *Some a* ⊕ *Some b* ⊕ *Some c*
  shows *b* ## *c*
  by (*metis assms compatible-eq option.discI plus.simps*(*2*) *plus-asso*)

lemma *add-fh-map-empty*:
  add-fh h Map.empty = *h*
proof (*rule ext*)
  fix *x* **show** *add-fh h Map.empty x* = *h x*
    by (*metis add-fh-def fadd-options.simps*(*1*) *fadd-options.simps*(*2*) *not-None-eq*)
qed

definition *bounded* **where**
  bounded h ⟷ (∀ *l p*. *fst h l* = *Some p* ⟶ *pgte pwrite* (*fst p*))

lemma *boundedI*:
assumes $\bigwedge l\ p.\ fst\ h\ l = Some\ p \Longrightarrow pgte\ pwrite\ (fst\ p)$
shows *bounded h*
by (*simp add*: *assms bounded-def*)

lemma *boundedE*:
assumes *bounded h*
and $fst\ h\ l = Some\ p$
shows *pgte pwrite* (*fst p*)
by (*meson assms*(*1*) *assms*(*2*) *bounded-def*)

lemma *bounded-smaller-sum*:
assumes *bounded x*
and $Some\ x = Some\ a \oplus Some\ b$
shows *bounded a*
proof (*rule boundedI*)
fix *l p* **assume** *asm0*: $fst\ a\ l = Some\ p$
then obtain p' **where** $fst\ x\ l = Some\ p'$
by (*metis assms*(*2*) *get-fh.simps one-value-sum-same prod.collapse*)
then have *pgte* (*fst p'*) (*fst p*)
apply (*cases fst b l*)
apply (*metis* (*no-types*, *lifting*) *add-fh-def add-get-fh asm0 assms*(*2*) *fadd-options.simps*(*2*) *get-fh.elims not-pgte-charact option.sel pgt-implies-pgte*)
using *add-fh-def*[*of fst a fst b l*] *asm0 assms*(*2*) *fadd-options.elims*[*of fst a l fst b l*]
fst-eqD get-fh.simps option.discI option.sel pgt-implies-pgte plus.simps(*3*)[*of a b*] *sum-larger*[*of*]
by *metis*
then show *pgte pwrite* (*fst p*)
by (*meson* ‹$fst\ x\ l = Some\ p'$› *assms*(*1*) *boundedE dual-order.trans pgte.rep-eq*)
qed

lemma *bounded-smaller*:
assumes *bounded x*
and $x \succeq a$
shows *bounded a*
using *assms*(*1*) *assms*(*2*) *bounded-smaller-sum larger-def* **by** *blast*

lemma *sum-perm-smaller*:
assumes $Some\ x = Some\ a \oplus Some\ b$
and $fst\ a\ l = Some\ (p, v)$
shows $\exists\, p'.\ pgte\ p'\ p \wedge fst\ x\ l = Some\ (p', v)$
apply (*cases fst b l*)
apply (*metis assms*(*1*) *assms*(*2*) *get-fh.simps order.refl pgte.rep-eq sum-second-none-get-fh*)
apply *clarsimp*
using *assms*(*2*) *fst-eqD get-fh.simps pgt-implies-pgte plus-extract-point-fh*[*OF assms*(*1*)] *snd-eqD sum-larger*

  by *simp*

lemma *modus-ponens*:
  assumes *A*
      and $A \Longrightarrow B$
    shows *B*
  using *assms* **by** *simp*

lemma *fpdom-inclusion*:
  assumes $Some\ h' = Some\ h \oplus Some\ r$
      and *bounded* h'
  shows *fpdom* (*fst h*) $\subseteq$ *fpdom* (*fst* h')
  apply *rule*
  unfolding *fpdom-def* **apply** *simp*
  apply (*erule exE*)
  subgoal for *x v*
    apply (*rule modus-ponens*[*of* $\exists\, p.\ fst\ h'\ x = Some\ (p,\ v)$])
     apply (*metis assms*(*1*) *get-fh.simps one-value-sum-same*)
    apply (*erule exE*)
    subgoal for *p*
      apply (*rule modus-ponens*[*of pgte pwrite p*])
       apply (*metis assms*(*2*) *boundedE fst-conv*)
      apply (*rule modus-ponens*[*of pgte p pwrite*])
      apply (*metis Pair-inject assms*(*1*) *option.inject sum-perm-smaller*)
      using *pgte-antisym* **by** *blast*
    done
  done

lemma *fpdom-dom-disjoint*:
  assumes *Some h = Some h1* $\oplus$ *Some h2*
  shows *dom* (*fst h1*) $\cap$ *fpdom* (*fst h2*) = {}
  apply *rule*
   apply *rule*
  unfolding *dom-def fpdom-def* **apply** *simp-all*
  apply (*erule conjE*)
  apply (*erule exE*)+
   by (*metis* (*no-types*, *lifting*) *assms fst-eqD get-fh.simps not-pgte-charact plus-comm plus-extract-point-fh sum-larger*)

lemma *fpdom-dom-union*:
  assumes *Some h = Some h1* $\oplus$ *Some h2*
      and *bounded h*
  shows *fpdom* (*fst h1*) $\cup$ *fpdom* (*fst h2*) $\subseteq$ *fpdom* (*fst h*)
  by (*metis assms fpdom-inclusion plus-comm sup-least*)

lemma *full-ownership-then-bounded*:
  assumes *full-ownership* (*fst h*)

shows *bounded h*
apply (*rule boundedI*)
by (*metis assms full-ownership-def pgte.rep-eq pwrite.rep-eq rel-simps*(*47*))

end

2 Imperative Concurrent Language

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

theory *Lang*
imports *Main StateModel*
begin

2.1 Language Syntax and Semantics

type-synonym *state* = *store* × *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*]: $[\![\ \sigma = (s,h);\ \sigma' = (s,\ h(edenot\ E\ s := None))\ ]\!] \Longrightarrow red\ (Cdispose\ E)\ \sigma\ Cskip\ \sigma'$

| *NoStep*: $red\text{-}rtrans\ C\ \sigma\ C\ \sigma$
| *OneMoreStep*: $[\![\ red\ C\ \sigma\ C'\ \sigma'\ ;\ red\text{-}rtrans\ C'\ \sigma'\ C''\ \sigma''\ ]\!] \Longrightarrow red\text{-}rtrans\ C\ \sigma\ C''\ \sigma''$

inductive-cases *red-par-cases*: $red\ (Cpar\ C1\ C2)\ \sigma\ C'\ \sigma'$
inductive-cases *red-atomic-cases*: $red\ (Catomic\ C)\ \sigma\ C'\ \sigma'$

2.1.3 Abort semantics

primrec
  $accesses :: cmd \Rightarrow store \Rightarrow nat\ set$
where
  $accesses\ Cskip\ \ s = \{\}$
| $accesses\ (Cassign\ x\ E)\ \ s = \{\}$
| $accesses\ (Cread\ x\ E)\ \ s = \{edenot\ E\ s\}$
| $accesses\ (Cwrite\ E\ E')\ \ s = \{edenot\ E\ s\}$
| $accesses\ (Calloc\ x\ E)\ \ s = \{\}$
| $accesses\ (Cdispose\ E)\ \ s = \{edenot\ E\ s\}$
| $accesses\ (Cseq\ C1\ C2)\ \ s = accesses\ C1\ s$
| $accesses\ (Cpar\ C1\ C2)\ \ s = accesses\ C1\ s \cup accesses\ C2\ s$
| $accesses\ (Cif\ B\ C1\ C2)\ \ s = \{\}$
| $accesses\ (Cwhile\ B\ C)\ \ s = \{\}$
| $accesses\ (Catomic\ C)\ \ s = \{\}$

primrec
  $writes :: cmd \Rightarrow store \Rightarrow nat\ set$
where
  $writes\ Cskip\ \ s = \{\}$
| $writes\ (Cassign\ x\ E)\ \ s = \{\}$
| $writes\ (Cread\ x\ E)\ \ s = \{\}$
| $writes\ (Cwrite\ E\ E')\ \ s = \{edenot\ E\ s\}$
| $writes\ (Calloc\ x\ E)\ \ s = \{\}$
| $writes\ (Cdispose\ E)\ \ s = \{edenot\ E\ s\}$
| $writes\ (Cseq\ C1\ C2)\ \ s = writes\ C1\ s$
| $writes\ (Cpar\ C1\ C2)\ \ s = writes\ C1\ s \cup writes\ C2\ s$
| $writes\ (Cif\ B\ C1\ C2)\ \ s = \{\}$
| $writes\ (Cwhile\ B\ C)\ \ s = \{\}$
| $writes\ (Catomic\ C)\ \ s = \{\}$

inductive
  $aborts :: cmd \Rightarrow state \Rightarrow bool$
where
  aborts-Seq[*intro*]: $aborts\ C1\ \sigma \Longrightarrow aborts\ (Cseq\ C1\ C2)\ \sigma$

| *aborts-Atomic*[*intro*]: $[\![$ *red-rtrans* $C\ \sigma\ C'\ \sigma'$; *aborts* $C'\ \sigma'$ $]\!]$ $\Longrightarrow$ *aborts* (*Catomic* C) σ
| *aborts-Par1*[*intro*]: *aborts C1* $\sigma \Longrightarrow$ *aborts* (*Cpar C1 C2*) σ
| *aborts-Par2*[*intro*]: *aborts C2* $\sigma \Longrightarrow$ *aborts* (*Cpar C1 C2*) σ
| *aborts-Read*[*intro*]: *edenot E* (*fst* σ) $\notin$ *dom* (*snd* σ) $\Longrightarrow$ *aborts* (*Cread x E*) σ
| *aborts-Write*[*intro*]: *edenot E* (*fst* σ) $\notin$ *dom* (*snd* σ) $\Longrightarrow$ *aborts* (*Cwrite E E'*) σ
| *aborts-Free*[*intro*]: *edenot E* (*fst* σ) $\notin$ *dom* (*snd* σ) $\Longrightarrow$ *aborts* (*Cdispose E*) σ
| *aborts-Race1*[*intro*]: *accesses C1* (*fst* σ) $\cap$ *writes C2* (*fst* σ) $\neq \{\}$ $\Longrightarrow$ *aborts* (*Cpar C1 C2*) σ
| *aborts-Race2*[*intro*]: *writes C1* (*fst* σ) $\cap$ *accesses C2* (*fst* σ) $\neq \{\}$ $\Longrightarrow$ *aborts* (*Cpar C1 C2*) 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* $\Rightarrow$ *var set*
where
  fvE (*Evar v*) = {*v*}
| *fvE* (*Enum n*) = {}
| *fvE* (*Eplus e1 e2*) = (*fvE e1* $\cup$ *fvE e2*)

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

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

primrec
  $wrC :: cmd \Rightarrow var\ set$
where
  $wrC\ (Cskip) = \{\}$
| $wrC\ (Cassign\ v\ E) = \{v\}$
| $wrC\ (Cread\ v\ E) = \{v\}$
| $wrC\ (Cwrite\ E1\ E2) = \{\}$
| $wrC\ (Calloc\ v\ E) = \{v\}$
| $wrC\ (Cdispose\ E) = \{\}$
| $wrC\ (Cseq\ C1\ C2) = (wrC\ C1 \cup wrC\ C2)$
| $wrC\ (Cpar\ C1\ C2) = (wrC\ C1 \cup wrC\ C2)$
| $wrC\ (Cif\ B\ C1\ C2) = (wrC\ C1 \cup wrC\ C2)$
| $wrC\ (Cwhile\ B\ C) = (wrC\ C)$
| $wrC\ (Catomic\ C) = (wrC\ C)$

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

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

Basic properties of substitutions:

lemma *subE-assign*:
 $edenot\ (subE\ x\ E\ e)\ s = edenot\ e\ (s(x := edenot\ E\ s))$
  by (*induct e*, *simp-all*)

lemma *subB-assign*:
 $bdenot\ (subB\ x\ E\ b)\ s = bdenot\ b\ (s(x := edenot\ E\ s))$
proof (*induct b*)
  case (*Beq x1 x2*)
  then show *?case*
    using *bdenot.simps*(*1*) *subB.simps*(*1*) *subE-assign* **by** *presburger*
qed (*simp-all*)

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* σ C' σ'
$\neg$ *aborts Cskip* σ
using *red-skip-cases* **apply** *blast*
using *aborts-skip-cases* **by** *blast*

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* $A\ s\ s' \land$ *agrees* $B\ s\ s'$
by (*meson Un-iff agrees-def*)

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

lemma *agreesI*:
assumes $\bigwedge x.\ x \in X \Longrightarrow s\ x = s'\ x$
shows *agrees* $X\ s\ s'$
using *agrees-def assms* **by** *blast*

lemma *red-properties*:
red $C\ \sigma\ C'\ \sigma' \Longrightarrow$ *fvC* $C' \subseteq$ *fvC* $C \land$ *wrC* $C' \subseteq$ *wrC* $C \land$ *agrees* $(-$ *wrC* $C)$ (*fst* σ') (*fst* σ)
red-rtrans $C\ \sigma\ C'\ \sigma' \Longrightarrow$ *fvC* $C' \subseteq$ *fvC* $C \land$ *wrC* $C' \subseteq$ *wrC* $C \land$ *agrees* $(-$ *wrC* $C)$ (*fst* σ') (*fst* σ)
proof (*induct rule*: *red-red-rtrans.inducts*)
case (*OneMoreStep* $C\ \sigma\ C'\ \sigma'\ C''\ \sigma''$)
then have *fvC* $C'' \subseteq$ *fvC* C
by *blast*
moreover have *wrC* $C'' \subseteq$ *wrC* C
using *OneMoreStep.hyps*(*2*) *OneMoreStep.hyps*(*4*) **by** *blast*
moreover have *agrees* $(-$ *wrC* $C)$ (*fst* σ'') (*fst* σ)
proof (*rule agreesI*)
fix x **assume** $x \in -$ *wrC* C
then have $x \in -$*wrC* $C' \land x \in -$*wrC* C''
using *OneMoreStep.hyps*(*2*) *OneMoreStep.hyps*(*4*) **by** *blast*
then show *fst* $\sigma''\ x =$ *fst* $\sigma\ x$
by (*metis OneMoreStep.hyps*(*2*) *OneMoreStep.hyps*(*4*) ‹$x \in -$ *wrC* C› *agrees-def*)
qed
ultimately show *?case* **by** *simp*
qed (*auto simp add*: *agrees-def*)

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* $E\ s'$
by (*simp add*: *agrees-def*, *induct* E, *auto*)

lemma *bexp-agrees*:

agrees (*fvB B*) *s s*′ $\Longrightarrow$ *bdenot B s* = *bdenot B s*′
proof (*induct B*)
case (*Beq x1 x2*)
then have *agrees* (*fvE x1*) *s s*′ $\wedge$ *agrees* (*fvE x2*) *s s*′
by (*simp add*: *agrees-def*)
then show *?case* **using** *exp-agrees*
by *force*
next
case (*Band B1 B2*)
then show *?case*
by (*simp add*: *agrees-def*)
qed (*simp-all*)

lemma *red-not-in-fv-not-touched*:
red C σ *C*′ σ' $\Longrightarrow$ $x \notin$ *fvC C* $\Longrightarrow$ *fst* σ *x* = *fst* σ' *x*
red-rtrans C σ *C*′ σ' $\Longrightarrow$ $x \notin$ *fvC C* $\Longrightarrow$ *fst* σ *x* = *fst* σ' *x*
proof (*induct arbitrary*: *rule*: *red-red-rtrans.inducts*)
case (*OneMoreStep C* σ *C*′ σ' *C*′′ σ'')
then show *fst* σ *x* = *fst* σ'' *x*
by (*metis red-properties*(*1*) *subsetD*)
qed (*auto*)

lemma *agrees-update1*:
assumes *agrees X s s*′
shows *agrees X* (*s*(*x* := *v*)) (*s*′(*x* := *v*))
proof (*rule agreesI*)
fix *y* **show** $y \in X$ $\Longrightarrow$ (*s*(*x* := *v*)) *y* = (*s*′(*x* := *v*)) *y*
apply (*cases y* = *x*)
apply *simp*
using *agrees-def assms* **by** *fastforce*
qed

lemma *agrees-update2*:
assumes *agrees X s s*′
and $x \notin X$
shows *agrees X* (*s*(*x* := *v*)) (*s*′(*x* := *v*′))
proof (*rule agreesI*)
fix *y* **show** $y \in X$ $\Longrightarrow$ (*s*(*x* := *v*)) *y* = (*s*′(*x* := *v*′)) *y*
apply (*cases y* = *x*)
using *assms*(*2*) **apply** *blast*
using *agrees-def assms*(*1*) **by** *fastforce*
qed

lemma *red-agrees-aux*:
red C σ *C*′ σ' $\Longrightarrow$ ($\forall$ *s h*. *agrees X* (*fst* σ) *s* $\wedge$ *snd* σ = *h* $\wedge$ *fvC C* $\subseteq$ *X* $\longrightarrow$
($\exists$ *s*′ *h*′. *red C* (*s*, *h*) *C*′ (*s*′, *h*′) $\wedge$ *agrees X* (*fst* σ') *s*′ $\wedge$ *snd* σ' = *h*′))
red-rtrans C σ *C*′ σ' $\Longrightarrow$ ($\forall$ *s h*. *agrees X* (*fst* σ) *s* $\wedge$ *snd* σ = *h* $\wedge$ *fvC C* $\subseteq$ *X*
$\longrightarrow$
($\exists$ *s*′ *h*′. *red-rtrans C* (*s*, *h*) *C*′ (*s*′, *h*′) $\wedge$ *agrees X* (*fst* σ') *s*′ $\wedge$ *snd* σ' = *h*′))

proof (*induct rule*: *red-red-rtrans.inducts*)
  case (*red-If1 B* σ *C1 C2*)
  then show *?case*
  proof (*clarify*)
    fix *X s h*
    assume *asm0*: *bdenot B* (*fst* σ) *agrees X* (*fst* σ) *s fvC* (*Cif B C1 C2*) $\subseteq$ *X*
    then have *bdenot B s*
      using *Un-iff agrees-def*[*of X fst* σ *s*] *bexp-agrees fvC.simps*(*9*) *in-mono agrees-def*[*of fvB B*]
      by *fastforce*
    then show $\exists s'\ h'$. *red* (*Cif B C1 C2*) (*s*, *snd* σ) *C1* (s', h') $\wedge$ *agrees X* (*fst* σ) $s' \wedge$ *snd* $\sigma = h'$
      by (*metis asm0*(*2*) *fst-eqD red-red-rtrans.red-If1*)
  qed
next
  case (*red-If2 B* σ *C1 C2*)
  then show *?case*
  proof (*clarify*)
    fix *X s h*
    assume *asm0*: $\neg$ *bdenot B* (*fst* σ) *agrees X* (*fst* σ) *s fvC* (*Cif B C1 C2*) $\subseteq$ *X*
    then have $\neg$ *bdenot B s*
    using *Un-subset-iff agrees-def*[*of X*] *agrees-def*[*of fvB B*] *bexp-agrees fvC.simps*(*9*) *in-mono*
      by *metis*
    then show $\exists s'\ h'$. *red* (*Cif B C1 C2*) (*s*, *snd* σ) *C2* (s', h') $\wedge$ *agrees X* (*fst* σ) $s' \wedge$ *snd* $\sigma = h'$
      by (*metis asm0*(*2*) *fst-eqD red-red-rtrans.red-If2*)
  qed
next
  case (*red-Assign* σ *ss hh* σ' *x E*)
  then show *?case*
  proof (*clarify*)
    fix *X s h*
    assume *asm0*: $\sigma' = (ss(x := \mathit{edenot}\ E\ ss), hh)$ $\sigma = (ss, hh)$ *agrees X* (*fst* (*ss*, *hh*)) *s fvC* (*Cassign x E*) $\subseteq$ *X*
    then have *edenot E s* = *edenot E ss*
      using *exp-agrees fst-conv fvC.simps*(*2*)
      by (*metis* (*mono-tags*, *lifting*) *Un-subset-iff agrees-def in-mono*)
    then have *red* (*Cassign x E*) (*ss*, *snd* (*s*, *h*)) *Cskip* ($ss(x := \mathit{edenot}\ E\ s)$, *h*)
      by *force*
    moreover have *agrees X* (*fst* ($s(x := \mathit{edenot}\ E\ s)$, *h*)) ($ss(x := \mathit{edenot}\ E\ s)$)
    proof (*rule agreesI*)
      fix *y* **assume** $y \in X$
      show *fst* ($s(x := \mathit{edenot}\ E\ s)$, *h*) *y* = ($ss(x := \mathit{edenot}\ E\ s)$) *y*
        apply (*cases* $x = y$)
        apply *simp*
        by (*metis* ‹$y \in X$› *agrees-def asm0*(*3*) *fstI fun-upd-other*)
    qed
    ultimately show $\exists s'\ h'$. *red* (*Cassign x E*) (*s*, *snd* (*ss*, *hh*)) *Cskip* (s', h') $\wedge$

agrees X (*fst* (*ss*(*x* := *edenot E ss*), *hh*)) *s'* ∧ *snd* (*ss*(*x* := *edenot E ss*), *hh*) = *h'*
using ‹*edenot E s* = *edenot E ss*›
by (*metis agrees-update1 asm0*(*3*) *fst-conv red-red-rtrans.red-Assign snd-conv*)
qed
next
case (*red-Read σ ss hh E v σ' x*)
have ⋀*s h*. *agrees X* (*fst σ*) *s* ∧ *snd σ* = *h* ∧ *fvC* (*Cread x E*) ⊆ *X* ⟹ (∃ *s' h'*. *red* (*Cread x E*) (*s*, *h*) *Cskip* (*s'*, *h'*) ∧ *agrees X* (*fst σ'*) *s'* ∧ *snd σ'* = *h'*)
proof −
fix *s h* **assume** *asm0*: *agrees X* (*fst σ*) *s* ∧ *snd σ* = *h* ∧ *fvC* (*Cread x E*) ⊆ *X*
then have *hh* (*edenot E s*) = *Some v*
using *red-Read*(*1*) *red-Read*(*2*) *exp-agrees fstI fvC.simps*(*3*) *Un-subset-iff agrees-def*[*of fvE E*] *in-mono*
agrees-def[*of X*] **by** *metis*
then have *agrees X* (*fst σ'*) (*s*(*x* := *v*))
by (*metis asm0*(*1*) *agrees-update1 fstI red-Read.hyps*(*1*) *red-Read.hyps*(*3*) *red-Read.prems*)
then show ∃ *s' h'*. *red* (*Cread x E*) (*s*, *h*) *Cskip* (*s'*, *h'*) ∧ *agrees X* (*fst σ'*) *s'* ∧ *snd σ'* = *h'*
using ‹*hh* (*edenot E s*) = *Some v*› *red-Read.hyps*(*1*) *red-Read.hyps*(*3*) *red-Read.prems*
by (*metis asm0 red-red-rtrans.red-Read snd-conv*)
qed
then show *?case* **by** *blast*
next
case (*red-Write σ ss hh σ' E E'*)
have ⋀*s h*. *agrees X* (*fst σ*) *s* ∧ *snd σ* = *h* ∧ *fvC* (*Cwrite E E'*) ⊆ *X* ⟹ (∃ *s' h'*. *red* (*Cwrite E E'*) (*s*, *h*) *Cskip* (*s'*, *h'*) ∧ *agrees X* (*fst σ'*) *s'* ∧ *snd σ'* = *h'*)
proof −
fix *s h* **assume** *asm0*: *agrees X* (*fst σ*) *s* ∧ *snd σ* = *h* ∧ *fvC* (*Cwrite E E'*) ⊆ *X*
then have *edenot E ss* = *edenot E s* ∧ *edenot E' ss* = *edenot E' s*
using *red-Write*(*1*) *exp-agrees fstI fvC.simps*(*4*)
by (*metis* (*mono-tags*, *lifting*) *Un-subset-iff agrees-def in-mono*)
then show ∃ *s' h'*. *red* (*Cwrite E E'*) (*s*, *h*) *Cskip* (*s'*, *h'*) ∧ *agrees X* (*fst σ'*) *s'* ∧ *snd σ'* = *h'*
by (*metis fst-conv asm0 red-Write.hyps*(*1*) *red-Write.hyps*(*2*) *red-Write.prems red-red-rtrans.red-Write snd-conv*)
qed
then show *?case* **by** *blast*
next
case (*red-Alloc σ ss hh v σ' x E*)
have ⋀*s h*. *agrees X* (*fst σ*) *s* ∧ *snd σ* = *h* ∧ *fvC* (*Calloc x E*) ⊆ *X* ⟹ (∃ *s' h'*. *red* (*Calloc x E*) (*s*, *h*) *Cskip* (*s'*, *h'*) ∧ *agrees X* (*fst σ'*) *s'* ∧ *snd σ'* = *h'*)
proof −
fix *s h* **assume** *asm0*: *agrees X* (*fst σ*) *s* ∧ *snd σ* = *h* ∧ *fvC* (*Calloc x E*) ⊆ *X*
then have *edenot E ss* = *edenot E s*
using *red-Alloc*(*1*) *exp-agrees fst-conv fvC.simps*(*5*)
by (*metis* (*mono-tags*, *lifting*) *Un-iff agrees-def in-mono*)

then have *agrees* X (*fst* σ') ($s(x := v)$)
by (*metis agrees-update1 asm0 fstI red-Alloc.hyps*(*1*) *red-Alloc.hyps*(*3*) *red-Alloc.prems*)
then show $\exists s'\ h'$. *red* (*Calloc x E*) (s, h) *Cskip* (s', h') $\wedge$ *agrees* X (*fst* σ') s' $\wedge$ *snd* $\sigma' = h'$
by (*metis* ‹*edenot E ss* = *edenot E s*› *red-Alloc.hyps*(*1*) *red-Alloc.hyps*(*2*) *red-Alloc.hyps*(*3*) *red-Alloc.prems red-red-rtrans.red-Alloc snd-eqD asm0*)
qed
then show *?case* **by** *blast*
next
case (*red-Free* σ *ss hh* σ' *E*)
have $\bigwedge s\ h$. *agrees* X (*fst* σ) s $\wedge$ *snd* $\sigma = h$ $\wedge$ *fvC* (*Cdispose E*) $\subseteq X \Longrightarrow$ ($\exists s'\ h'$. *red* (*Cdispose E*) (s, h) *Cskip* (s', h') $\wedge$ *agrees* X (*fst* σ') s' $\wedge$ *snd* $\sigma' = h'$)
proof −
fix s h **assume** *asm0*: *agrees* X (*fst* σ) s $\wedge$ *snd* $\sigma = h$ $\wedge$ *fvC* (*Cdispose E*) $\subseteq X$
then have *edenot E ss* = *edenot E s*
using *red-Free*(*1*) *exp-agrees fst-eqD fvC.simps*(*6*)
by (*metis agrees-def in-mono*)
then show $\exists s'\ h'$. *red* (*Cdispose E*) (s, h) *Cskip* (s', h') $\wedge$ *agrees* X (*fst* σ') s' $\wedge$ *snd* $\sigma' = h'$
using *red-Free.hyps*(*1*) *red-Free.hyps*(*2*) *red-Free.prems asm0* **by** *fastforce*
qed
then show *?case* **by** *blast*
next
case (*NoStep C* σ)
then show *?case*
using *red-red-rtrans.NoStep* **by** *blast*
next
case (*OneMoreStep C* σ C' σ' C'' σ'')
have $\bigwedge s\ h$. *agrees* X (*fst* σ) s $\wedge$ *snd* $\sigma = h$ $\wedge$ *fvC* $C \subseteq X \Longrightarrow$ ($\exists s'\ h'$. *red-rtrans* C (s, h) C'' (s', h') $\wedge$ *agrees* X (*fst* σ'') s' $\wedge$ *snd* $\sigma'' = h'$)
proof −
fix s h **assume** *asm0*: *agrees* X (*fst* σ) s $\wedge$ *snd* $\sigma = h$ $\wedge$ *fvC* $C \subseteq X$
then obtain h' s' **where** *red* C (s, h) C' (s', h') $\wedge$ *agrees* X (*fst* σ') s' $\wedge$ *snd* $\sigma' = h'$
using *OneMoreStep*(*2*) **by** *auto*
then obtain h'' s'' **where** *red-rtrans* C' (s', h') C'' (s'', h'') $\wedge$ *agrees* X (*fst* σ'') s'' $\wedge$ *snd* $\sigma'' = h''$
using *OneMoreStep.hyps*(*4*) *asm0 red-properties*(*1*) **by** *fastforce*
then show $\exists s'\ h'$. *red-rtrans* C (s, h) C'' (s', h') $\wedge$ *agrees* X (*fst* σ'') s' $\wedge$ *snd* $\sigma'' = h'$
using ‹*red* C (s, h) C' (s', h') $\wedge$ *agrees* X (*fst* σ') s' $\wedge$ *snd* $\sigma' = h'$› *red-red-rtrans.OneMoreStep* **by** *blast*
qed
then show *?case* **by** *blast*
qed (*fastforce*+)

lemma *red-agrees*[*rule-format*]:
red C σ C' σ' $\Longrightarrow \forall X\ s$. *agrees* X (*fst* σ) 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* σ') s' $\wedge$ *snd* $\sigma' = h'$)

using *red-agrees-aux*(*1*) **by** *blast*

lemma *writes-accesses*: *writes C s* $\subseteq$ *accesses C s*
by (*induct C arbitrary*: *s*, *auto*)

lemma *accesses-agrees*: *agrees* (*fvC C*) *s s'* $\Longrightarrow$ *accesses C s* = *accesses C s'*
apply (*induct C arbitrary*: *s s'*)
apply (*simp-all add*: *exp-agrees agrees-def*)
by *blast*

lemma *writes-agrees*: *agrees* (*fvC C*) *s s'* $\Longrightarrow$ *writes C s* = *writes C s'*
apply (*induct C arbitrary*: *s s'*)
apply (*simp-all add*: *exp-agrees agrees-def*)
by *blast*

lemma *aborts-agrees*:
assumes *aborts C* σ
and *agrees* (*fvC C*) (*fst* σ) *s*
and *snd* σ = *h*
shows *aborts C* (*s*, *h*)
using *assms*
proof (*induct arbitrary*: *s h rule*: *aborts.induct*)
case (*aborts-Atomic C* σ *C'* σ')
then obtain *s'* **where** *red-rtrans C* (*s*, *h*) *C'* (*s'*, *snd* σ') $\wedge$ *agrees* (*fvC C*) (*fst* σ') *s'*
by (*metis dual-order.refl fvC.simps*(*11*) *red-agrees-aux*(*2*))
moreover have *agrees* (*fvC C'*) (*fst* σ') *s'*
using *calculation red-properties*(*2*)
by (*meson agrees-def in-mono*)
ultimately show *?case*
using *aborts-Atomic.hyps*(*3*) **by** *blast*
next
case (*aborts-Read E* σ *x*)
then show *?case*
using *aborts.aborts-Read*[*of E* σ *x*] *exp-agrees*[*of E fst* σ *s*] *fst-conv fvC.simps*(*3*) *snd-conv*
by (*simp add*: *aborts.aborts-Read agrees-def*)
next
case (*aborts-Write E* σ *E'*)
then show *?case*
using *aborts.aborts-Write*[*of E* σ *E'*] *exp-agrees*[*of - fst* σ *s*] *fst-conv fvC.simps*(*4*)[*of E E'*] *snd-conv*
by (*simp add*: *aborts.aborts-Write agrees-def*)
next
case (*aborts-Free E* σ)
then show *?case*
using *exp-agrees* **by** *auto*

next
  case (*aborts-Par1 C1* σ *C2*)
  then have *agrees* (*fvC C1*) (*fst* σ) *s*
    by (*simp add*: *agrees-def*)
  then show *?case* **using** *aborts.aborts-Par1*
    by (*simp add*: *aborts-Par1.hyps*(*2*) *aborts-Par1.prems*(*2*))
next
  case (*aborts-Par2 C2* σ *C1*)
  then have *agrees* (*fvC C2*) (*fst* σ) *s*
    by (*simp add*: *agrees-def*)
  then show *?case* **using** *aborts.aborts-Par2*
    by (*simp add*: *aborts-Par2.hyps*(*2*) *aborts-Par2.prems*(*2*))
next
  case (*aborts-Seq C1* σ *C2*)
  then have *agrees* (*fvC C1*) (*fst* σ) *s*
    by (*simp add*: *agrees-def*)
  then show *?case*
    by (*simp add*: *aborts.aborts-Seq aborts-Seq.hyps*(*2*) *aborts-Seq.prems*(*2*))
next
  case (*aborts-Race1 C1* σ *C2*)
  then show *?case* **using** *accesses-agrees*[*of C1 fst* σ *s*] *writes-agrees*[*of C2 fst* σ *s*] *aborts.aborts-Race1*[*of C1* (*s*, *h*) *C2*]
    by (*simp add*: *agrees-union*)
next
  case (*aborts-Race2 C1* σ *C2*)
  then show *?case* **using** *accesses-agrees*[*of C2 fst* σ *s*] *writes-agrees*[*of C1 fst* σ *s*] *aborts.aborts-Race2*[*of C1* (*s*, *h*) *C2*]
    by (*simp add*: *agrees-union*)
qed

corollary *exp-agrees2*[*simp*]:
  $x \notin fvE\ E \Longrightarrow edenot\ E\ (s(x := v)) = edenot\ E\ s$
by (*rule exp-agrees*, *simp add*: *agrees-def*)

lemma *agrees-update*:
  assumes $a \notin S$
  shows *agrees S s* $(s(a := v))$
  by (*simp add*: *agrees-def assms*)

lemma *agrees-comm*:
  $agrees\ S\ s\ s' \longleftrightarrow agrees\ S\ s'\ s$
  by (*metis agrees-def*)

lemma *not-in-dom*:
  assumes $x \notin dom\ s$
  shows $s\ x = None$
  using *assms* **by** *auto*

lemma *agrees-minusD*:
agrees $(-X)$ *x y* $\Longrightarrow X \cap Y = \{\} \Longrightarrow$ *agrees Y x y*
by (*metis Int-Un-eq*(*2*) *agrees-union compl-le-swap1 inf.order-iff inf-shunt*)

end

3 CommCSL

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

theory *CommCSL*
imports *Lang StateModel*
begin

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

typedef $'a$ *precondition* = { *pre* :: ($'a \Rightarrow 'a \Rightarrow$ *bool*) |*pre*. $\forall$ *a b*. *pre a b* $\longrightarrow$ (*pre b a* $\wedge$ *pre a a*) }
using *Sup2-E* **by** *auto*

lemma *charact-rep-prec*:
assumes *Rep-precondition pre a b*
shows *Rep-precondition pre b a* $\wedge$ *Rep-precondition pre a a*
using *Rep-precondition assms* **by** *fastforce*

typedef ($'i$, $'a$) *indexed-precondition* = { *pre* :: ($'i \Rightarrow 'a \Rightarrow 'a \Rightarrow$ *bool*) |*pre*. $\forall$ *a b k*. *pre k a b* $\longrightarrow$ (*pre k b a* $\wedge$ *pre k a a*) }
using *Sup2-E* **by** *auto*

lemma *charact-rep-indexed-prec*:
assumes *Rep-indexed-precondition pre k a b*
shows *Rep-indexed-precondition pre k b a* $\wedge$ *Rep-indexed-precondition pre k a a*
by (*metis* (*no-types*, *lifting*) *Abs-indexed-precondition-cases Rep-indexed-precondition-cases Rep-indexed-precondition-inverse assms mem-Collect-eq*)

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

3.1 Assertion Language

datatype ($'i$, $'a$, $'v$) *assertion* =
Bool bexp
| *Emp*
| *And* ($'i$, $'a$, $'v$) *assertion* ($'i$, $'a$, $'v$) *assertion*
| *Star* ($'i$, $'a$, $'v$) *assertion* ($'i$, $'a$, $'v$) *assertion* (‹- ∗ -› *70*)
| *Low bexp*
| *LowExp exp*

| *PointsTo exp prat exp*

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

| *EmptyFullGuards*

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

| *View normal-heap* $\Rightarrow$ *'v ('i, 'a, 'v) assertion store* $\Rightarrow$ *'v*
| *SharedGuard prat store* $\Rightarrow$ *'a multiset*
| *UniqueGuard 'i 'a list-exp*

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

inductive *PRE-shared-simpler* :: (*'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*) $\Rightarrow$ *'a multiset* $\Rightarrow$ *'a multiset* $\Rightarrow$ *bool* **where**
Empty: *PRE-shared-simpler spre* {#} {#}
| *Step*: $[\![$ *PRE-shared-simpler spre a b* ; *spre xa xb* $]\!] \Longrightarrow$ *PRE-shared-simpler spre* (*a* + {# *xa* #}) (*b* + {# *xb* #})

definition *PRE-unique* :: (*'b* $\Rightarrow$ *'b* $\Rightarrow$ *bool*) $\Rightarrow$ *'b list* $\Rightarrow$ *'b list* $\Rightarrow$ *bool* **where**
PRE-unique upre uargs uargs' $\longleftrightarrow$ *length uargs* = *length uargs'* $\wedge$ ($\forall$ *i*. *i* $\geq$ *0* $\wedge$ *i* < *length uargs'* $\longrightarrow$ *upre* (*uargs* ! *i*) (*uargs'* ! *i*))

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

fun *hyper-sat* :: (*store* $\times$ (*'i*, *'a*) *heap*) $\Rightarrow$ (*store* $\times$ (*'i*, *'a*) *heap*) $\Rightarrow$ (*'i*, *'a*, *nat*) *assertion* $\Rightarrow$ *bool* (‹-, - $\models$ -› [*51*, *65*, *66*] *50*) **where**
(*s*, -), (*s'*, -) $\models$ *Bool b* $\longleftrightarrow$ *bdenot b s* $\wedge$ *bdenot b s'*
| (-, *h*), (-, *h'*) $\models$ *Emp* $\longleftrightarrow$ *dom* (*get-fh h*) = {} $\wedge$ *dom* (*get-fh h'*) = {}
| σ, σ' $\models$ *And A B* $\longleftrightarrow$ σ, σ' $\models$ *A* $\wedge$ σ, σ' $\models$ *B*

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

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

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

| (*s*, *h*), (*s'*, *h'*) $\models$ *PreSharedGuards spre* $\longleftrightarrow$

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

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

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

$|\ (s,\ h),\ (s',\ h') \models LowExp\ e \longleftrightarrow edenot\ e\ s = edenot\ e\ s'$

$|\ (s,\ h),\ (s',\ h') \models Imp\ b\ A \longleftrightarrow bdenot\ b\ s = bdenot\ b\ s' \wedge (bdenot\ b\ s \longrightarrow (s,\ h),\ (s',\ h') \models A)$

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

lemma *sat-PreUniqueE*:
assumes $(s,\ h),\ (s',\ h') \models PreUniqueGuards\ upre$
shows $\exists\, uargs\ uargs'.\ (\forall\, k.\ get\text{-}gu\ h\ k = Some\ (uargs\ k)) \wedge (\forall\, k.\ get\text{-}gu\ h'\ k = Some\ (uargs'\ k)) \wedge (\forall\, k.\ PRE\text{-}unique\ (Rep\text{-}indexed\text{-}precondition\ upre\ k)\ (uargs\ k)\ (uargs'\ k))$
using *assms* **by** *auto*

lemma *decompose-heap-triple*:
$h = (get\text{-}fh\ h,\ get\text{-}gs\ h,\ get\text{-}gu\ h)$
by *simp*

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

lemma *depends-only-onI*:
assumes $\bigwedge s\ s' :: store.\ agrees\ S\ s\ s' \Longrightarrow e\ s = e\ s'$
shows *depends-only-on e S*
using *assms depends-only-on-def* **by** *blast*

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

fvS e = (*SOME S*. *depends-only-on e S*)

lemma *fvSE*:
assumes *agrees* (*fvS e*) *s s'*
shows *e s* = *e s'*
proof −
have *depends-only-on e UNIV*
proof (*rule depends-only-onI*)
fix *s s'* :: *store* **assume** *agrees UNIV s s'*
have *s* = *s'*
proof (*rule ext*)
fix *x* :: *var* **have** $x \in$ *UNIV*
by *auto*
then show *s x* = *s' x*
by (*meson* ‹*agrees UNIV s s'*› *agrees-def*)
qed
then show *e s* = *e s'* **by** *simp*
qed
then show *?thesis*
by (*metis assms depends-only-on-def fvS-def someI-ex*)
qed

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 ⟷ *e* (*s*(*x* := *edenot E s*))
by (*simp add*: *subS-def*)

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* ⟷ (*s*(*x* := *edenot E s*), *h*), (*s'*(*x* := *edenot E s'*), *h'*) ⊨ *A*
using *assms*
proof (*induct A arbitrary*: *s h s' h'*)
case (*And A1 A2*)
then show *?case*
by (*simp add*: *disjoint-iff-not-equal*)
next
case (*Star A1 A2*)
then show *?case*
by (*simp add*: *disjoint-iff-not-equal*)
next
case (*Bool x*)
then show *?case*
by (*metis hyper-sat.simps*(*1*) *subA.simps*(*3*) *subB-assign*)
next
case (*Low x2*)
then show *?case*
by (*metis* (*no-types*, *lifting*) *hyper-sat.simps*(*5*) *subA.simps*(*4*) *subB-assign*)
next
case (*LowExp x2*)
then show *?case*
by (*metis* (*no-types*, *lifting*) *hyper-sat.simps*(*14*) *subA.simps*(*5*) *subE-assign*)
next
case (*PointsTo x1a x2 x3*)
then show *?case*
by (*metis* (*no-types*, *lifting*) *hyper-sat.simps*(*6*) *subA.simps*(*9*) *subE-assign*)

next
  case (*Exists y A*)
  then have *asm0*: *collect-existentials A* $\cap$ *fvE E* = {}
    by *auto*
  show *?case* (**is** *?A* $\longleftrightarrow$ *?B*)
  proof
    show *?A* $\Longrightarrow$ *?B*
    proof –
      assume *?A*
      show *?B*
      proof (*cases* $x = y$)
        case *True*
        then show *?thesis* **by** (*metis* (*no-types*, *opaque-lifting*) ‹*?A*› *fun-upd-upd hyper-sat.simps*(*7*) *subA.simps*(*10*))
      next
        case *False*
        then obtain v v' **where** $(s(y := v), h), (s'(y := v'), h') \models$ *subA x E A*
          using ‹$(s, h), (s', h') \models$ *subA x E* (*Exists y A*)› **by** *auto*
        then have $((s(y := v))(x :=$ *edenot E* $(s(y := v))), h), ((s'(y := v'))(x :=$ *edenot E* $(s'(y := v'))), h') \models A$
          using *Exists asm0* **by** *blast*
        moreover have $y \notin$ *fvE E*
          using *Exists.prems* **by** *force*
        then have *edenot E* $(s(y := v)) =$ *edenot E s* $\wedge$ *edenot E* $(s'(y := v')) =$ *edenot E s'*
          by (*metis agrees-update exp-agrees*)
        moreover have $(s(y := v))(x :=$ *edenot E s*$) = (s(x :=$ *edenot E s*$))(y := v)$
        $\wedge$ $(s'(y := v'))(x :=$ *edenot E s'*$) = (s'(x :=$ *edenot E s'*$))(y := v')$
          by (*simp add*: *False fun-upd-twist*)
        ultimately show *?thesis* **using** *False hyper-sat.simps*(*7*)
          by *metis*
      qed
    qed
    assume *?B*
    show *?A*
    proof (*cases* $x = y$)
      case *True*
      then show *?thesis*
        using ‹$(s(x :=$ *edenot E s*$), h), (s'(x :=$ *edenot E s'*$), h') \models$ *Exists y A*› **by** *fastforce*
    next
      case *False*
      then obtain v v' **where** $((s(x :=$ *edenot E s*$))(y := v), h), ((s'(x :=$ *edenot E s'*$))(y := v'), h') \models A$
        using ‹$(s(x :=$ *edenot E s*$), h), (s'(x :=$ *edenot E s'*$), h') \models$ *Exists y A*› *hyper-sat.simps*(*7*) **by** *blast*
      moreover have $(s(y := v))(x :=$ *edenot E s*$) = (s(x :=$ *edenot E s*$))(y := v)$
      $\wedge$ $(s'(y := v'))(x :=$ *edenot E s'*$) = (s'(x :=$ *edenot E s'*$))(y := v')$

by (*simp add*: *False fun-upd-twist*)
then have $((s(y := v))(x := \textit{edenot } E\ (s(y := v))),\ h),\ ((s'(y := v'))(x := \textit{edenot } E\ (s'(y := v'))),\ h') \models A$
using *Exists.prems calculation* **by** *auto*
then show *?thesis*
by (*metis Exists.hyps False asm0 hyper-sat.simps*(*7*) *subA.simps*(*10*))
qed
qed
next
case (*View x1a A x3*)
then show *?case*
by (*metis* (*mono-tags*, *lifting*) *collect-existentials.simps*(*4*) *hyper-sat.simps*(*11*) *subA.simps*(*8*) *subS-def*)
next
case (*SharedGuard x1a x2*)
then show *?case*
by (*metis* (*mono-tags*, *lifting*) *hyper-sat.simps*(*12*) *subA.simps*(*7*) *subS-def*)
next
case (*UniqueGuard x*)
then show *?case*
by (*metis* (*mono-tags*, *lifting*) *hyper-sat.simps*(*13*) *subA.simps*(*6*) *subS-def*)
next
case (*Imp b A*)
then show *?case*
by (*metis collect-existentials.simps*(*5*) *hyper-sat.simps*(*15*) *subA.simps*(*11*) *subB-assign*)
qed (*auto*)

lemma *PRE-uniqueI*:
assumes *length uargs* = *length uargs'*
and $\bigwedge i.\ i \geq 0 \wedge i < \textit{length uargs}' \Longrightarrow \textit{upre}\ (\textit{uargs} ! i)\ (\textit{uargs}' ! i)$
shows *PRE-unique upre uargs uargs'*
using *assms PRE-unique-def* **by** *blast*

lemma *PRE-unique-implies-tl*:
assumes *PRE-unique upre* (*ta* # *qa*) (*tb* # *qb*)
shows *PRE-unique upre qa qb*
proof (*rule PRE-uniqueI*)
show *length qa* = *length qb*
by (*metis PRE-unique-def assms diff-Suc-1 length-Cons*)
fix *i* **assume** $0 \leq i \wedge i < \textit{length qb}$
then have *upre* ((*ta* # *qa*) ! (*i* + *1*)) ((*tb* # *qb*) ! (*i* + *1*))
using *assms PRE-unique-def* [*of upre* ‹*ta* # *qa*› ‹*tb* # *qb*›]
by (*auto simp add*: *less-Suc-eq-le dest*: *spec* [*of* - ‹*Suc i*›])
then show *upre* (*qa* ! *i*) (*qb* ! *i*)
by *simp*
qed

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*
using *assms*
proof (*induct length a arbitrary*: *a b*)
case *0*
then show *?case*
by (*simp add*: *PRE-unique-def*)
next
case (*Suc n*)
then obtain *ha ta hb tb* **where** *a* = *ha* # *ta b* = *hb* # *tb*
by (*metis One-nat-def PRE-unique-def Suc-le-length-iff le-add1 plus-1-eq-Suc*)
then have *n* = *length ta*
using *Suc.hyps*(*2*) **by** *auto*
then have *PRE-unique* (*Rep-indexed-precondition pre k*) *tb ta* ∧ *PRE-unique* (*Rep-indexed-precondition pre k*) *ta ta*
by (*metis PRE-unique-implies-tl Suc.hyps*(*1*) *Suc.prems* ‹*a* = *ha* # *ta*› ‹*b* = *hb* # *tb*›)
then show *?case*
by (*metis PRE-unique-def Suc.prems charact-rep-indexed-prec*)
qed

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*
using *assms*
proof (*induct arbitrary*: *pre rule*: *PRE-shared-simpler.induct*)
case (*Empty spre*)
then show *?case*
by (*simp add*: *PRE-shared-simpler.Empty*)
next
case (*Step spre a b xa xb*)
then have *spre xb xa* ∧ *spre xa xa* **using** *charact-rep-prec* **by** *metis*
then show *?case*
by (*metis PRE-shared-simpler.Step Step.hyps*(*2*) *Step.prems*)
qed

lemma *always-sat-refl-aux*:
assumes $(s, h), (s', h') \models A$
shows $(s, h), (s, h) \models A$
using *assms*
proof (*induct A arbitrary*: $s\ h\ s'\ h'$)
case (*Star A B*)
then obtain $ha\ hb\ ha'\ hb'$ **where** $Some\ h = Some\ ha \oplus Some\ hb$ $Some\ h' = Some\ ha' \oplus Some\ hb'$

$(s, ha), (s', ha') \models A$ $(s, hb), (s', hb') \models B$
by *auto*
then have $(s, ha), (s, ha) \models A \wedge (s, hb), (s, hb) \models B$
using *Star.hyps*(*1*) *Star.hyps*(*2*) **by** *blast*
then show *?case*
using ‹*Some h* = *Some ha* ⊕ *Some hb*› *hyper-sat.simps*(*4*) **by** *blast*
next
case (*Exists x A*)
then show *?case*
by (*meson hyper-sat.simps*(*7*))
next
case (*PreSharedGuards x*)
then show *?case*
using *charact-PRE-shared-simpler* **by** *force*
next
case (*PreUniqueGuards upre*)
then obtain *uargs uargs'* **where** (∀ *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*
using *hyper-sat.simps*(*10*)[*of s h s' h' upre*] **by** *blast*
then show $(s, h), (s, h) \models$ *PreUniqueGuards upre*
using *charact-PRE-unique hyper-sat.simps*(*10*)[*of s h s h upre*]
by *metis*
qed (*auto*)

lemma *always-sat-refl*:
assumes $\sigma, \sigma' \models A$
shows $\sigma, \sigma \models A$
by (*metis always-sat-refl-aux assms prod.exhaust-sel*)

lemma *agrees-same-aux*:
assumes *agrees* (*fvA A*) *s s''*
and $(s, h), (s', h') \models A$
shows $(s'', h), (s', h') \models A$
using *assms*
proof (*induct A arbitrary*: *s s' s'' h h'*)
case (*Bool b*)
then show *?case* **by** (*simp add*: *bexp-agrees*)
next
case (*And A1 A2*)
then show *?case* **using** *fvA.simps*(*2*) *hyper-sat.simps*(*3*)
by (*metis* (*mono-tags*, *lifting*) *UnCI agrees-def*)
next
case (*Star A B*)
then obtain *ha hb ha' hb'* **where** *Some h* = *Some ha* ⊕ *Some hb* *Some h'* = *Some ha'* ⊕ *Some hb'*
$(s, ha), (s', ha') \models A$ $(s, hb), (s', hb') \models B$
by *auto*
then have $(s'', ha), (s', ha') \models A \wedge (s'', hb), (s', hb') \models B$

using *Star.hyps*[*of s s''* - *s'* -] *Star.prems*(*1*)
by (*simp add*: *agrees-def*)
then show *?case*
using ‹*Some h* = *Some ha* ⊕ *Some hb*› ‹*Some h'* = *Some ha'* ⊕ *Some hb'*› *hyper-sat.simps*(*4*) **by** *blast*
next
case (*Low e*)
then have *bdenot e s* = *bdenot e s''*
by (*metis bexp-agrees fvA.simps*(*4*))
then show *?case* **using** *Low* **by** *simp*
next
case (*LowExp e*)
then have *edenot e s* = *edenot e s''*
by (*metis exp-agrees fvA.simps*(*14*))
then show *?case* **using** *LowExp* **by** *simp*
next
case (*PointsTo l* π *v*)
then have *edenot l s* = *edenot l s''* ∧ *edenot v s* = *edenot v s''*
by (*simp add*: *agrees-def exp-agrees*)
then show *?case* **using** *PointsTo* **by** *auto*
next
case (*Exists x A*)
then obtain *v v'* **where** $(s(x := v), h), (s'(x := v'), h') \models A$
by *auto*
moreover have *agrees* (*fvA A*) $(s(x := v))$ $(s''(x := v))$
proof (*rule agreesI*)
fix *y* **show** $y \in fvA\ A \Longrightarrow (s(x := v))\ y = (s''(x := v))\ y$
apply (*cases y* = *x*)
apply *simp*
using *Diff-iff*[*of y fvA A* {*x*}] *Exists.prems*(*1*) *agrees-def empty-iff fun-upd-apply*[*of s x v*] *fun-upd-apply*[*of s'' x v*] *fvA.simps*(*7*) *insert-iff*
by *metis*
qed
ultimately have $(s''(x := v), h), (s'(x := v'), h') \models A$
using *Exists.hyps* **by** *blast*
then show *?case* **by** *auto*
next
case (*View x1a A e*)
then have $(s'', h), (s', h') \models A \wedge e\ s = e\ s''$
using *fvA.simps*(*10*) *fvSE hyper-sat.simps*(*11*) *agrees-union*
by *metis*
then show *?case*
using *View.prems*(*2*) **by** *auto*
next
case (*SharedGuard x1a x2*)
then show *?case* **using** *fvSE* **by** *auto*
next
case (*UniqueGuard x*)
then show *?case* **using** *fvSE* **by** *auto*

next
  case $(Imp\ b\ A)$
  then show *?case*
    by $(metis\ agrees\text{-}union\ bexp\text{-}agrees\ fvA.simps(15)\ hyper\text{-}sat.simps(15))$
qed $(auto)$

lemma *agrees-same*:
  assumes $agrees\ (fvA\ A)\ s\ s''$
  shows $(s,\ h),\ (s',\ h') \models A \longleftrightarrow (s'',\ h),\ (s',\ h') \models A$
  by $(metis\ (mono\text{-}tags,\ lifting)\ agrees\text{-}def\ agrees\text{-}same\text{-}aux\ assms)$

lemma *sat-comm-aux*:
  $(s,\ h),\ (s',\ h') \models A \Longrightarrow (s',\ h'),\ (s,\ h) \models A$
proof $(induct\ A\ arbitrary:\ s\ h\ s'\ h')$
  case $(Star\ A\ B)$
  then obtain $ha\ hb\ ha'\ hb'$ **where** $Some\ h = Some\ ha \oplus Some\ hb\ Some\ h' = Some\ ha' \oplus Some\ hb'$
    $(s,\ ha),\ (s',\ ha') \models A\ (s,\ hb),\ (s',\ hb') \models B$
    by *auto*
  then have $(s',\ ha'),\ (s,\ ha) \models A \wedge (s',\ hb'),\ (s,\ hb) \models B$
    using $Star.hyps(1)\ Star.hyps(2)$ **by** *presburger*
  then show *?case*
    using ‹$Some\ h = Some\ ha \oplus Some\ hb$› ‹$Some\ h' = Some\ ha' \oplus Some\ hb'$› $hyper\text{-}sat.simps(4)$ **by** *blast*
next
  case $(Exists\ x\ A)$
  then obtain $v\ v'$ **where** $(s(x := v),\ h),\ (s'(x := v'),\ h') \models A$
    by *auto*
  then have $(s'(x := v'),\ h'),\ (s(x := v),\ h) \models A$
    using *Exists.hyps* **by** *blast*
  then show *?case* **by** *auto*
next
  case $(PreSharedGuards\ x)$
  then show *?case*
    by $(meson\ charact\text{-}PRE\text{-}shared\text{-}simpler\ hyper\text{-}sat.simps(9))$
next
  case $(PreUniqueGuards\ upre)$
  then obtain $uargs\ uargs'$ **where** $(\forall k.\ get\text{-}gu\ h\ k = Some\ (uargs\ k)) \wedge$
    $(\forall k.\ get\text{-}gu\ h'\ k = Some\ (uargs'\ k)) \wedge (\forall k.\ PRE\text{-}unique\ (Rep\text{-}indexed\text{-}precondition\ upre\ k)\ (uargs\ k)\ (uargs'\ k)) \wedge get\text{-}fh\ h = Map.empty \wedge get\text{-}fh\ h' = Map.empty$
    using $hyper\text{-}sat.simps(10)[of\ s\ h\ s'\ h'\ upre]$ **by** *blast*
  then show $(s',\ h'),\ (s,\ h) \models PreUniqueGuards\ upre$
    using $charact\text{-}PRE\text{-}unique\ hyper\text{-}sat.simps(10)[of\ s'\ h'\ s\ h\ upre]$
    by *metis*
qed $(auto)$

lemma *sat-comm*:
  $\sigma,\ \sigma' \models A \longleftrightarrow \sigma',\ \sigma \models A$
  using *sat-comm-aux surj-pair* **by** *metis*

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

lemma *preciseI*:
assumes $\bigwedge s1\ H1\ h1\ h1'\ s2\ H2\ h2\ h2'.\ H1 \succeq h1 \wedge H1 \succeq h1' \wedge H2 \succeq h2 \wedge H2 \succeq h2' \Longrightarrow (s1, h1), (s2, h2) \models J \Longrightarrow (s1, h1'), (s2, h2') \models J \Longrightarrow h1' = h1 \wedge h2' = h2$
shows *precise* J
using *assms precise-def* **by** *blast*

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 $h1' = h1 \wedge h2' = h2$
using *assms*(1) *assms*(2) *assms*(3) *precise-def* **by** *blast*

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
using *assms unary-def* **by** *blast*

lemma *unary-smallerI*:
assumes $\bigwedge \sigma 1\ \sigma 2.\ \sigma 1, \sigma 1 \models J \wedge \sigma 2, \sigma 2 \models J \Longrightarrow \sigma 1, \sigma 2 \models J$
shows *unary* J
using *assms unary-def* **by** *blast*

lemma *unaryE*:
assumes *unary* J
and $(s, h), (s, h) \models J \wedge (s', h'), (s', h') \models J$
shows $(s, h), (s', h') \models J$
using *assms*(1) *assms*(2) *unary-def* **by** *blast*

definition *entails* :: $('i, 'a, nat)$ *assertion* $\Rightarrow$ $('i, 'a, nat)$ *assertion* $\Rightarrow$ *bool* **where**
entails $A\ B \longleftrightarrow (\forall \sigma\ \sigma'.\ \sigma, \sigma' \models A \longrightarrow \sigma, \sigma' \models B)$

lemma *entailsI*:

assumes $\bigwedge x\ y.\ x,\ y \models A \Longrightarrow x,\ y \models B$
shows *entails A B*
using *assms entails-def* **by** *blast*

lemma *sat-points-to*:
assumes $(s,\ h :: ('i,\ 'a)\ heap),\ (s,\ h) \models PointsTo\ a\ \pi\ e$
shows $get\text{-}fh\ h = [edenot\ a\ s \mapsto (\pi,\ edenot\ e\ s)]$
proof −
have $get\text{-}fh\ h\ (edenot\ a\ s) = Some\ (\pi,\ edenot\ e\ s) \wedge dom\ (get\text{-}fh\ h) = \{edenot\ a\ s\}$
using *assms* **by** *auto*
then show *?thesis*
by *fastforce*
qed

lemma *unary-inv-then-view*:
assumes *unary J*
shows *unary* (*View f J e*)
proof (*rule unaryI*)
fix $s\ h\ s'\ h'$
assume *asm0*: $(s,\ h),\ (s,\ h) \models View\ f\ J\ e \wedge (s',\ h'),\ (s',\ h') \models View\ f\ J\ e$
then show $(s,\ h),\ (s',\ h') \models View\ f\ J\ e$
by (*meson assms hyper-sat.simps*(*11*) *unaryE*)
qed

lemma *precise-inv-then-view*:
assumes *precise J*
shows *precise* (*View f J e*)
proof (*rule preciseI*)
fix $s1\ H1\ h1\ h1'\ s2\ H2\ h2\ h2'$
assume *asm0*: $H1 \succeq h1 \wedge H1 \succeq h1' \wedge H2 \succeq h2 \wedge H2 \succeq h2'\ (s1,\ h1),\ (s2,\ h2) \models View\ f\ J\ e$
$(s1,\ h1'),\ (s2,\ h2') \models View\ f\ J\ e$
then show $h1' = h1 \wedge h2' = h2$
by (*meson assms hyper-sat.simps*(*11*) *preciseE*)
qed

fun *syntactic-unary* :: $('i,\ 'a,\ nat)\ assertion \Rightarrow bool$ **where**
syntactic-unary (*Bool b*) $\longleftrightarrow$ *True*
| *syntactic-unary* (*And A B*) $\longleftrightarrow$ *syntactic-unary A* $\wedge$ *syntactic-unary B*
| *syntactic-unary* (*Star A B*) $\longleftrightarrow$ *syntactic-unary A* $\wedge$ *syntactic-unary B*
| *syntactic-unary* (*Low e*) $\longleftrightarrow$ *False*
| *syntactic-unary Emp* $\longleftrightarrow$ *True*
| *syntactic-unary* (*PointsTo v va vb*) $\longleftrightarrow$ *True*
| *syntactic-unary* (*Exists x A*) $\longleftrightarrow$ *syntactic-unary A*
| *syntactic-unary* (*SharedGuard - e*) $\longleftrightarrow$ *True*
| *syntactic-unary* (*UniqueGuard - e*) $\longleftrightarrow$ *True*

| *syntactic-unary* (*View view A e*) $\longleftrightarrow$ *syntactic-unary A*
| *syntactic-unary* (*PreSharedGuards* -) $\longleftrightarrow$ *False*
| *syntactic-unary* (*PreUniqueGuards* -) $\longleftrightarrow$ *False*
| *syntactic-unary EmptyFullGuards* $\longleftrightarrow$ *True*
| *syntactic-unary* (*LowExp x*) $\longleftrightarrow$ *False*
| *syntactic-unary* (*Imp b A*) $\longleftrightarrow$ *False*

lemma *syntactic-unary-implies-unary*:
  assumes *syntactic-unary A*
  shows *unary A*
  using *assms*
proof (*induct A*)
  case (*And A1 A2*)
  show *?case*
  proof (*rule unary-smallerI*)
    fix *σ1 σ2*
    assume *σ1*, *σ1* ⊨ *And A1 A2* ∧ *σ2*, *σ2* ⊨ *And A1 A2*
    then have *σ1*, *σ2* ⊨ *A1* ∧ *σ1*, *σ2* ⊨ *A2*
      using *And unary-def*
      by (*metis hyper-sat.simps*(*3*) *prod.exhaust-sel syntactic-unary.simps*(*2*))
    then show *σ1*, *σ2* ⊨ *And A1 A2*
      using *hyper-sat.simps*(*3*) **by** *blast*
  qed
next
  case (*Star A B*)
  then have *unary A* ∧ *unary B* **by** *simp*
  show *?case*
  proof (*rule unaryI*)
    fix *s1 h1 s2 h2*
    assume *asm0*: (*s1*, *h1*), (*s1*, *h1*) ⊨ *Star A B* ∧ (*s2*, *h2*), (*s2*, *h2*) ⊨ *Star A B*
    then obtain *a1 b1 a2 b2* **where** *Some h1* = *Some a1* ⊕ *Some b1* (*s1*, *a1*), (*s1*, *a1*) ⊨ *A* (*s1*, *b1*), (*s1*, *b1*) ⊨ *B*
      Some h2 = *Some a2* ⊕ *Some b2* (*s2*, *a2*), (*s2*, *a2*) ⊨ *A* (*s2*, *b2*), (*s2*, *b2*) ⊨ *B*
      by (*meson always-sat-refl hyper-sat.simps*(*4*))
    then have (*s1*, *a1*), (*s2*, *a2*) ⊨ *A* ∧ (*s1*, *b1*), (*s2*, *b2*) ⊨ *B*
      using ‹*unary A* ∧ *unary B*› *unaryE* **by** *blast*
    then show (*s1*, *h1*), (*s2*, *h2*) ⊨ *Star A B*
      using ‹*Some h1* = *Some a1* ⊕ *Some b1*› ‹*Some h2* = *Some a2* ⊕ *Some b2*› *hyper-sat.simps*(*4*) **by** *blast*
  qed
next
  case (*Exists x A*)
  then have *unary A* **by** *simp*
  show *?case*
  proof (*rule unaryI*)
    fix *s1 h1 s2 h2*
    assume (*s1*, *h1*), (*s1*, *h1*) ⊨ *Exists x A* ∧ (*s2*, *h2*), (*s2*, *h2*) ⊨ *Exists x A*

then obtain *v1 v2* **where** $(s1(x := v1), h1), (s1(x := v1), h1) \models A \wedge (s2(x := v2), h2), (s2(x := v2), h2) \models A$
by (*meson always-sat-refl hyper-sat.simps*(*7*))
then have $(s1(x := v1), h1), (s2(x := v2), h2) \models A$
using ‹*unary A*› *unary-def* **by** *blast*
then show $(s1, h1), (s2, h2) \models$ *Exists x A* **by** *auto*
qed
next
case (*View view A x*)
then have *unary A* **by** *simp*
show *?case*
proof (*rule unaryI*)
fix *s1 h1 s2 h2*
assume *asm0*: $(s1, h1), (s1, h1) \models$ *View view A x* $\wedge$ $(s2, h2), (s2, h2) \models$ *View view A x*
then have $(s1, h1), (s2, h2) \models A$
by (*meson* ‹*unary A*› *hyper-sat.simps*(*11*) *unaryE*)
then show $(s1, h1), (s2, h2) \models$ *View view A x*
using *asm0* **by** *fastforce*
qed
qed (*auto simp add*: *unary-def*)

The following record defines resource contexts (Section 3.5).

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*. $(s1, h1), (s2, h2) \models A$ ⟶ *no-guard h1* ∧ *no-guard h2*)

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

definition *view-function-of-inv* :: (*'i*, *'a*, *nat*) *single-context* ⇒ *bool* **where**
view-function-of-inv Γ ⟷ (∀ (*h* :: (*'i*, *'a*) *heap*) (*h'* :: (*'i*, *'a*) *heap*) *s*. $(s, h), (s, h) \models$ *invariant* Γ ∧ ($h' \succeq h$)
⟶ *view* Γ (*normalize* (*get-fh h*)) = *view* Γ (*normalize* (*get-fh h'*)))

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

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

lemma *wf-precondition-rep-prec*:
  assumes *wf-precondition pre*
  shows *Rep-precondition* (*Abs-precondition pre*) = *pre*
 using *Abs-precondition-inverse*[*of pre*] *assms mem-Collect-eq wf-precondition-def*[*of pre*]
  by *blast*

lemma *wf-indexed-precondition-rep-prec*:
  assumes *wf-indexed-precondition pre*
  shows *Rep-indexed-precondition* (*Abs-indexed-precondition pre*) = *pre*
 using *Abs-indexed-precondition-inverse*[*of pre*] *assms mem-Collect-eq wf-indexed-precondition-def*[*of pre*]
  by *blast*

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

lemma *LowViewE*:
  assumes $(s, h), (s', h') \models LowView\ f\ A\ x$
    and $x \notin fvA\ A$
   shows $(s, h), (s', h') \models A \wedge f\ (normalize\ (get\text{-}fh\ h)) = f\ (normalize\ (get\text{-}fh\ h'))$
proof −
  obtain $v\ v'$ **where** $(s(x := v), h), (s'(x := v'), h') \models And\ (View\ f\ A\ (\lambda s.\ s\ x))\ (LowExp\ (Evar\ x))$
   by (*metis LowView-def assms*(*1*) *hyper-sat.simps*(*7*))
  then obtain $(s(x := v), h), (s'(x := v'), h') \models View\ f\ A\ (\lambda s.\ s\ x)$
     $(s(x := v), h), (s'(x := v'), h') \models LowExp\ (Evar\ x)$
   using *hyper-sat.simps*(*3*) **by** *blast*
  then obtain $(s(x := v), h), (s'(x := v'), h') \models A$ $v = v'$
    $f\ (normalize\ (get\text{-}fh\ h)) = f\ (normalize\ (get\text{-}fh\ h'))$
   by *simp*
  moreover have $(s, h), (s', h') \models A$
   by (*meson agrees-same agrees-update assms*(*2*) *calculation*(*1*) *sat-comm-aux*)
  ultimately show *?thesis* **by** *blast*
qed

lemma *LowViewI*:
  assumes $(s, h), (s', h') \models A$
    and $f\ (normalize\ (get\text{-}fh\ h)) = f\ (normalize\ (get\text{-}fh\ h'))$
    and $x \notin fvA\ A$
   shows $(s, h), (s', h') \models LowView\ f\ A\ x$
proof −
  let $?s = s(x := f\ (normalize\ (get\text{-}fh\ h)))$
  let $?s' = s'(x := f\ (normalize\ (get\text{-}fh\ h')))$
  have $(?s, h), (?s', h') \models A$

by (*meson agrees-same-aux agrees-update assms*(*1*) *assms*(*3*) *sat-comm-aux*)
then have (*?s*, *h*), (*?s'*, *h'*) $\models$ *And* (*View f A* ($\lambda s.$ *s x*)) (*LowExp* (*Evar x*))
using *assms*(*2*) **by** *auto*
then show *?thesis* **using** *LowView-def*
by (*metis hyper-sat.simps*(*7*))
qed

definition *disjoint* :: (*'a set*) $\Rightarrow$ (*'a set*) $\Rightarrow$ *bool*
where *disjoint h1 h2* = (*h1* $\cap$ *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:
($\forall$ *v v' sarg sarg'*. α *v* = α *v'* $\wedge$ *spre sarg sarg'* $\longrightarrow$ α (*sact v sarg*) = α (*sact v' sarg'*)) $\wedge$
($\forall$ *v v' uarg uarg' i*. α *v* = α *v'* $\wedge$ *upre i uarg uarg'* $\longrightarrow$ α (*uact i v uarg*) = α (*uact i v' uarg'*)) $\wedge$

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

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

3.2 Rules of the Logic

inductive *CommCSL* :: (*'i*, *'a*, *nat*) *cont* $\Rightarrow$ (*'i*, *'a*, *nat*) *assertion* $\Rightarrow$ *cmd* $\Rightarrow$ (*'i*, *'a*, *nat*) *assertion* $\Rightarrow$ *bool*
(‹- $\vdash$ {-} - {-}› [*51*,*0*,*0*] *81*) **where**
RuleSkip: $\Delta \vdash$ {*P*} *Cskip* {*P*}
| *RuleAssign*: $[\![$ $\bigwedge\Gamma$. Δ = *Some* Γ $\Longrightarrow$ *x* $\notin$ *fvA* (*invariant* Γ) ; *collect-existentials P* $\cap$ *fvE E* = {} $]\!]$ $\Longrightarrow$ $\Delta \vdash$ {*subA x E P*} *Cassign x E* {*P*}
| *RuleNew*: $[\![$ *x* $\notin$ *fvE E*; $\bigwedge\Gamma$. Δ = *Some* Γ $\Longrightarrow$ *x* $\notin$ *fvA* (*invariant* Γ) $\wedge$ *view-function-of-inv* Γ $]\!]$ $\Longrightarrow$ $\Delta \vdash$ {*Emp*} *Calloc x E* {*PointsTo* (*Evar x*) *pwrite E*}
| *RuleWrite*: $[\![$ $\bigwedge\Gamma$. Δ = *Some* Γ $\Longrightarrow$ *view-function-of-inv* Γ ; *v* $\notin$ *fvE loc* $]\!]$

$\Longrightarrow \Delta \vdash \{Exists\ v\ (PointsTo\ loc\ pwrite\ (Evar\ v))\}\ Cwrite\ loc\ E\ \{PointsTo\ loc\ pwrite\ E\}$
$|\ \llbracket\ \bigwedge\Gamma.\ \Delta = Some\ \Gamma \Longrightarrow x \notin fvA\ (invariant\ \Gamma) \wedge view\text{-}function\text{-}of\text{-}inv\ \Gamma\ ;\ x \notin fvE\ E \cup fvE\ e\ \rrbracket \Longrightarrow$
$\Delta \vdash \{PointsTo\ E\ \pi\ e\}\ Cread\ x\ E\ \{And\ (PointsTo\ E\ \pi\ e)\ (Bool\ (Beq\ (Evar\ x)\ e))\}$
$|\ RuleShare$: $\llbracket\ \Gamma = (\!|\ view = f,\ abstract\text{-}view = \alpha,\ saction = sact,\ uaction = uact,\ invariant = J\ |\!)\ ;\ all\text{-}axioms\ \alpha\ sact\ spre\ uact\ upre\ ;$
$Some\ \Gamma \vdash \{Star\ P\ EmptyFullGuards\}\ C\ \{Star\ Q\ (And\ (PreSharedGuards\ (Abs\text{-}precondition\ spre))\ (PreUniqueGuards\ (Abs\text{-}indexed\text{-}precondition\ upre)))\}$;
$view\text{-}function\text{-}of\text{-}inv\ \Gamma\ ;\ unary\ J\ ;\ precise\ J\ ;\ wf\text{-}indexed\text{-}precondition\ upre\ ;\ wf\text{-}precondition\ spre\ ;\ x \notin fvA\ J\ ;$
$no\text{-}guard\text{-}assertion\ (Star\ P\ (LowView\ (\alpha \circ f)\ J\ x))\ \rrbracket \Longrightarrow None \vdash \{Star\ P\ (LowView\ (\alpha \circ f)\ J\ x)\}\ C\ \{Star\ Q\ (LowView\ (\alpha \circ f)\ J\ x)\}$
$|\ RuleAtomicUnique$: $\llbracket\ \Gamma = (\!|\ view = f,\ abstract\text{-}view = \alpha,\ saction = sact,\ uaction = uact,\ invariant = J\ |\!)\ ;$
$no\text{-}guard\text{-}assertion\ P\ ;\ no\text{-}guard\text{-}assertion\ Q\ ;$
$None \vdash \{Star\ P\ (View\ f\ J\ (\lambda s.\ s\ x))\}\ C\ \{Star\ Q\ (View\ f\ J\ (\lambda s.\ uact\ index\ (s\ x)\ (map\text{-}to\text{-}arg\ (s\ uarg))))\}\ ;$
$precise\ J\ ;\ unary\ J\ ;\ view\text{-}function\text{-}of\text{-}inv\ \Gamma\ ;\ x \notin fvC\ C \cup fvA\ P \cup fvA\ Q \cup fvA\ J\ ;\ uarg \notin fvC\ C\ ;$
$l \notin fvC\ C\ ;\ x \notin fvS\ (\lambda s.\ map\text{-}to\text{-}list\ (s\ l))\ ;\ x \notin fvS\ (\lambda s.\ map\text{-}to\text{-}arg\ (s\ uarg)\ \#\ map\text{-}to\text{-}list\ (s\ l))\ \rrbracket$
$\Longrightarrow Some\ \Gamma \vdash \{Star\ P\ (UniqueGuard\ index\ (\lambda s.\ map\text{-}to\text{-}list\ (s\ l)))\}\ Catomic\ C\ \{Star\ Q\ (UniqueGuard\ index\ (\lambda s.\ map\text{-}to\text{-}arg\ (s\ uarg)\ \#\ map\text{-}to\text{-}list\ (s\ l)))\}$
$|\ RuleAtomicShared$: $\llbracket\ \Gamma = (\!|\ view = f,\ abstract\text{-}view = \alpha,\ saction = sact,\ uaction = uact,\ invariant = J\ |\!)\ ;\ no\text{-}guard\text{-}assertion\ P\ ;\ no\text{-}guard\text{-}assertion\ Q\ ;$
$None \vdash \{Star\ P\ (View\ f\ J\ (\lambda s.\ s\ x))\}\ C\ \{Star\ Q\ (View\ f\ J\ (\lambda s.\ sact\ (s\ x)\ (map\text{-}to\text{-}arg\ (s\ sarg))))\}\ ;$
$precise\ J\ ;\ unary\ J\ ;\ view\text{-}function\text{-}of\text{-}inv\ \Gamma\ ;\ x \notin fvC\ C \cup fvA\ P \cup fvA\ Q \cup fvA\ J\ ;\ sarg \notin fvC\ C\ ;$
$ms \notin fvC\ C\ ;\ x \notin fvS\ (\lambda s.\ map\text{-}to\text{-}multiset\ (s\ ms))\ ;\ x \notin fvS\ (\lambda s.\ \{\#\ map\text{-}to\text{-}arg\ (s\ sarg)\ \#\} + map\text{-}to\text{-}multiset\ (s\ ms))\ \rrbracket$
$\Longrightarrow Some\ \Gamma \vdash \{Star\ P\ (SharedGuard\ \pi\ (\lambda s.\ map\text{-}to\text{-}multiset\ (s\ ms)))\}\ Catomic\ C\ \{Star\ Q\ (SharedGuard\ \pi\ (\lambda s.\ \{\#\ map\text{-}to\text{-}arg\ (s\ sarg)\ \#\} + map\text{-}to\text{-}multiset\ (s\ ms)))\}$
$|\ RulePar$: $\llbracket\ \Delta \vdash \{P1\}\ C1\ \{Q1\}\ ;\ \Delta \vdash \{P2\}\ C2\ \{Q2\}\ ;\ disjoint\ (fvA\ P1 \cup fvC\ C1 \cup fvA\ Q1)\ (wrC\ C2)\ ;$
$disjoint\ (fvA\ P2 \cup fvC\ C2 \cup fvA\ Q2)\ (wrC\ C1)\ ;\ \bigwedge\Gamma.\ \Delta = Some\ \Gamma \Longrightarrow disjoint\ (fvA\ (invariant\ \Gamma))\ (wrC\ C2)\ ;$
$\bigwedge\Gamma.\ \Delta = Some\ \Gamma \Longrightarrow disjoint\ (fvA\ (invariant\ \Gamma))\ (wrC\ C1)\ ;\ precise\ P1 \vee precise\ P2\ \rrbracket$
$\Longrightarrow \Delta \vdash \{Star\ P1\ P2\}\ Cpar\ C1\ C2\ \{Star\ Q1\ Q2\}$
$|\ RuleIf1$: $\llbracket\ \Delta \vdash \{And\ P\ (Bool\ b)\}\ C1\ \{Q\}\ ;\ \Delta \vdash \{And\ P\ (Bool\ (Bnot\ b))\}\ C2\ \{Q\}\ \rrbracket$
$\Longrightarrow \Delta \vdash \{And\ P\ (Low\ b)\}\ Cif\ b\ C1\ C2\ \{Q\}$
$|\ RuleIf2$: $\llbracket\ \Delta \vdash \{And\ P\ (Bool\ b)\}\ C1\ \{Q\}\ ;\ \Delta \vdash \{And\ P\ (Bool\ (Bnot\ b))\}\ C2\ \{Q\}\ ;\ unary\ Q\ \rrbracket$
$\Longrightarrow \Delta \vdash \{P\}\ Cif\ b\ C1\ C2\ \{Q\}$

| *RuleSeq*: $\llbracket\ \Delta \vdash \{P\}\ C1\ \{R\}\ ;\ \Delta \vdash \{R\}\ C2\ \{Q\}\ \rrbracket \Longrightarrow \Delta \vdash \{P\}\ Cseq\ C1\ C2\ \{Q\}$
| *RuleFrame*: $\llbracket\ \Delta \vdash \{P\}\ C\ \{Q\}\ ;\ disjoint\ (fvA\ R)\ (wrC\ C)\ ;\ precise\ P \vee precise\ R\ \rrbracket$
$\Longrightarrow \Delta \vdash \{Star\ P\ R\}\ C\ \{Star\ Q\ R\}$
| *RuleCons*: $\llbracket\ \Delta \vdash \{P'\}\ C\ \{Q'\}\ ;\ entails\ P\ P'\ ;\ entails\ Q'\ Q\ \rrbracket \Longrightarrow \Delta \vdash \{P\}\ C\ \{Q\}$
| *RuleExists*: $\llbracket\ \Delta \vdash \{P\}\ C\ \{Q\}\ ;\ x \notin fvC\ C\ ;\ \bigwedge\Gamma.\ \Delta = Some\ \Gamma \Longrightarrow x \notin fvA\ (invariant\ \Gamma)\ ;\ unambiguous\ P\ x\ \rrbracket$
$\Longrightarrow \Delta \vdash \{Exists\ x\ P\}\ C\ \{Exists\ x\ Q\}$
| *RuleWhile1*: $\Delta \vdash \{And\ I\ (Bool\ b)\}\ C\ \{And\ I\ (Low\ b)\} \Longrightarrow \Delta \vdash \{And\ I\ (Low\ b)\}\ Cwhile\ b\ C\ \{And\ I\ (Bool\ (Bnot\ b))\}$
| *RuleWhile2*: $\llbracket\ unary\ I\ ;\ \Delta \vdash \{And\ I\ (Bool\ b)\}\ C\ \{I\}\ \rrbracket \Longrightarrow \Delta \vdash \{I\}\ Cwhile\ b\ C\ \{And\ I\ (Bool\ (Bnot\ b))\}$

end

4 Soundness of CommCSL

4.1 Abstract Commutativity

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

theory *AbstractCommutativity*
imports *Main CommCSL HOL−Library.Multiset*
begin

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

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

lemma *sabstract*:
assumes *all-axioms* α *sact spre uact upre*
shows $\alpha\ v = \alpha\ v' \wedge spre\ sarg\ sarg' \Longrightarrow \alpha\ (sact\ v\ sarg) = \alpha\ (sact\ v'\ sarg')$
using *all-axioms-def*[*of* α *sact spre uact upre*] *assms* **by** *fast*

lemma *uabstract*:
assumes *all-axioms* α *sact spre uact upre*
shows $\alpha\ v = \alpha\ v' \wedge upre\ i\ uarg\ uarg' \Longrightarrow \alpha\ (uact\ i\ v\ uarg) = \alpha\ (uact\ i\ v'\ uarg')$
using *all-axioms-def*[*of* α *sact spre uact upre*] *assms* **by** *fast*

lemma *spre-refl*:
assumes *all-axioms* α *sact spre uact upre*
shows $spre\ sarg\ sarg' \Longrightarrow spre\ sarg'\ sarg'$
using *all-axioms-def*[*of* α *sact spre uact upre*] *assms* **by** *fast*

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

using *all-axioms-def*[*of* α *sact spre uact upre*] *assms* **by** *fast*

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*)
using *all-axioms-def*[*of* α *sact spre uact upre*] *assms* **by** *blast*

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*)
using *all-axioms-def*[*of* α *sact spre uact upre*] *assms* **by** *blast*

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 −
have $\bigwedge v\ v'$ *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*)
using *all-axioms-def*[*of* α *sact spre uact upre*] *assms*(*1*)
by *blast*
then show *?thesis*
using *assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) **by** *blast*
qed

definition *PRE-shared* :: ('*a* $\Rightarrow$ '*a* $\Rightarrow$ *bool*) $\Rightarrow$ '*a multiset* $\Rightarrow$ '*a multiset* $\Rightarrow$ *bool*
where
PRE-shared spre sargs sargs' $\longleftrightarrow$ ($\exists$ *ms*. *image-mset fst ms* = *sargs* $\wedge$ *image-mset snd ms* = *sargs'* $\wedge$ ($\forall x \in\#$ *ms*. *spre* (*fst x*) (*snd x*)))

lemma *PRE-shared-same-size*:
assumes *PRE-shared spre sargs sargs'*
shows *size sargs* = *size sargs'*
proof −
obtain *ms* **where** *image-mset fst ms* = *sargs* $\wedge$ *image-mset snd ms* = *sargs'* $\wedge$ ($\forall x \in\#$ *ms*. *spre* (*fst x*) (*snd x*))
by (*metis PRE-shared-def assms*)
then have *size sargs* = *size ms* $\wedge$ *size ms* = *size sargs'*
by *force*
then show *?thesis*
by *simp*
qed

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

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

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

definition *possible-sequence* :: $'a$ *multiset* $\Rightarrow ('i \Rightarrow 'b$ *list*$) \Rightarrow ('i, 'a, 'b)$ *action list* $\Rightarrow$ *bool* **where**
possible-sequence sargs uargs s $\longleftrightarrow ((\forall i.$ *uargs i* $=$ *map get-u* (*filter* (*is-Unique-i i*) *s*))
$\wedge$ *sargs* $=$ *image-mset get-s* (*filter-mset is-Shared* (*mset s*)))

lemma *possible-sequenceI*:
assumes $\bigwedge i.$ *uargs i* $=$ *map get-u* (*filter* (*is-Unique-i i*) *s*)
and *sargs* $=$ *image-mset get-s* (*filter-mset is-Shared* (*mset s*))
shows *possible-sequence sargs uargs s*
using *assms*(*1*) *assms*(*2*) *possible-sequence-def* **by** *blast*

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

lemma *remove-at-index*:
assumes $n <$ *length l*
shows *length* (*remove-at-index n l*) $=$ *length l* $- 1$
and $i \geq 0 \wedge i < n \Longrightarrow$ *remove-at-index n l* ! $i = l$! i
and $i \geq n \wedge i <$ *length l* $- 1 \Longrightarrow$ *remove-at-index n l* ! $i = l$! $(i + 1)$
using *assms*
proof (*induct l arbitrary*: *n i*)
case (*Cons a l*)
{
case *1*
then show *?case*
proof (*cases n*)
case (*Suc k*)
then show *?thesis*
using *1 Cons.hyps*(*1*) **by** *force*
qed (*simp*)
next
case *2*
then show *?case*
proof (*cases n*)
case (*Suc k*)
then show *?thesis*

```
        using 2.prems(1) 2.prems(2) Cons.hyps(2) Suc-less-eq2 less-Suc-eq-0-disj
by auto
    qed (simp)
  next
    case 3
    then show ?case
    proof (cases n)
      case (Suc k)
      then have remove-at-index (Suc k) (a # l) ! i = (a # l) ! (i + 1)
        apply (cases i)
        using 3.prems(1) apply blast
        using 3.prems(1) Cons.hyps(3) Suc-less-eq2 by auto
      then show remove-at-index n (a # l) ! i = (a # l) ! (i + 1)
        using Suc by blast
    qed (simp)
  }
qed (auto)

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 ≥ 0 ∧ i < n ⟹ insert-at n x l ! i = l ! i
    and n ≥ 0 ⟹ insert-at n x l ! n = x
    and i > n ∧ i < length l + 1 ⟹ insert-at n x l ! i = l ! (i − 1)
  using assms
proof (induct l arbitrary: n i)
  case (Cons a l)
  {
    case 1
    then show ?case by (cases n) (simp-all add: Cons.hyps(1))
  next
    case 2
    then show ?case apply (cases n)
       apply blast
      using Cons.hyps(2) less-Suc-eq-0-disj by force
  next
    case 3
    then show ?case apply (cases n)
       apply simp
      by (simp add: Cons.hyps(3))
  next
    case 4
    then show ?case apply (cases n)
       apply simp
```

by (*metis* (*no-types*, *lifting*) *4.prems*(*1*) *4.prems*(*2*) *Cons.hyps*(*4*) *Nat.add-0-right One-nat-def Suc-le-length-iff Suc-less-eq Suc-pred add-Suc-right bot-nat-0.not-eq-extremum insert-at.simps*(*3*) *less-zeroE list.inject list.size*(*4*) *nat.simps*(*3*) *nth-Cons′ nth-Cons-Suc*)
}
qed (*simp-all*)

lemma *list-ext*:
assumes $length\ a = length\ b$
and $\bigwedge i.\ i \geq 0 \wedge i < length\ a \Longrightarrow a\ !\ i = b\ !\ i$
shows $a = b$
by (*meson assms*(*1*) *assms*(*2*) *bot-nat-0.extremum nth-equalityI*)

lemma *mset-remove-index*:
assumes $i \geq 0 \wedge i < length\ l$
shows $mset\ l = mset\ (remove\text{-}at\text{-}index\ i\ l) + \{\#\ l\ !\ i\ \#\}$
using *assms*
proof (*induct l arbitrary*: *i*)
case (*Cons a l*)
then show *?case*
proof (*cases i*)
case (*Suc k*)
then show *?thesis*
using *Cons.hyps Cons.prems* **by** *force*
qed (*auto*)
qed (*simp*)

lemma *filter-remove*:
assumes $k \geq 0 \wedge k < length\ s$
and $\neg\ P\ (s\ !\ k)$
shows $filter\ P\ (remove\text{-}at\text{-}index\ k\ s) = filter\ P\ s$
using *assms*
proof (*induct k arbitrary*: *s*)
case *0*
then have $s = hd\ s\ \#\ tl\ s$
by *simp*
then show *?case*
by (*metis 0.prems*(*2*) *filter.simps*(*2*) *nth-Cons-0 remove-at-index.simps*(*2*))
next
case (*Suc k*)
then show *?case*
by (*cases s*) *simp-all*
qed

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

have *a* ∈# *image-mset get-s* (*filter-mset is-Shared* (*mset s*))
by (*metis assms*(*1*) *assms*(*2*) *possible-sequence-def*)
then have *Shared a* ∈ *set s*
by *fastforce*
then obtain *i* **where** $i \geq 0 \wedge i < \mathit{length}\ s \wedge s\ !\ i = \mathit{Shared}\ a$
by (*meson bot-nat-0.extremum in-set-conv-nth*)
let *?s* = *remove-at-index i s*
have *sargs* − {# *a* #} = *image-mset get-s* (*filter-mset is-Shared* (*mset ?s*))
proof −
have *sargs* = *image-mset get-s* (*filter-mset is-Shared* (*mset s*))
using *possible-sequence-def assms*(*2*) **by** *blast*
moreover have *mset s* = *mset ?s* + {# *Shared a* #}
by (*metis* ‹$0 \leq i \wedge i < \mathit{length}\ s \wedge s\ !\ i = \mathit{Shared}\ a$› *mset-remove-index*)
ultimately show *?thesis*
by *simp*
qed
moreover have ⋀*i*. *uargs i* = *map get-u* (*filter* (*is-Unique-i i*) *?s*)
by (*metis* ‹$0 \leq i \wedge i < \mathit{length}\ s \wedge s\ !\ i = \mathit{Shared}\ a$› *action.disc*(*1*) *assms*(*2*) *filter-remove is-Unique-def is-Unique-i-def possible-sequence-def*)
ultimately show *?thesis*
using ‹$0 \leq i \wedge i < \mathit{length}\ s \wedge s\ !\ i = \mathit{Shared}\ a$› *possible-sequence-def* **by** *blast*
qed

lemma *head-possible-exists-first-unique*:
assumes *a* = *hd* (*uargs j*)
and *uargs j* ≠ []
and *possible-sequence sargs uargs s*
shows $\exists i.\ i \geq 0 \wedge i < \mathit{length}\ s \wedge s\ !\ i = \mathit{Unique}\ j\ a \wedge (\forall k.\ k \geq 0 \wedge k < i \longrightarrow \neg\ \mathit{is\text{-}Unique\text{-}i}\ j\ (s\ !\ k))$
using *assms*
proof (*induct s arbitrary*: *sargs uargs*)
case *Nil*
then show *?case* **by** (*simp add*: *possible-sequence-def*)
next
case (*Cons x xs*)
then show $\exists i{\geq}0.\ i < \mathit{length}\ (x\ \#\ xs) \wedge (x\ \#\ xs)\ !\ i = \mathit{Unique}\ j\ a \wedge (\forall k.\ 0 \leq k \wedge k < i \longrightarrow \neg\ \mathit{is\text{-}Unique\text{-}i}\ j\ ((x\ \#\ xs)\ !\ k))$
proof (*cases x*)
case (*Shared sarg*)
moreover have *possible-sequence* (*sargs* − {# *sarg* #}) *uargs xs*
proof (*rule possible-sequenceI*)
show *sargs* − {#*sarg*#} = *image-mset get-s* (*filter-mset is-Shared* (*mset xs*))
using *Cons.prems*(*3*) *action.disc*(*1*) *action.sel*(*1*) *add-mset-remove-trivial*[*of sarg*]
calculation
filter-mset-add-mset image-mset-add-mset mset.simps(*2*) *possible-sequence-def*[*of sargs uargs x* # *xs*]
by *auto*
fix *i* **show** *uargs i* = *map get-u* (*filter* (*is-Unique-i i*) *xs*)

using *Cons.prems*(*3*) *action.disc*(*1*) *calculation filter-remove is-Unique-def is-Unique-i-def*
le-numeral-extra(*3*) *length-greater-0-conv list.discI nth-Cons-0 possible-sequence-def*[*of sargs uargs x # xs*]
remove-at-index.simps(*2*)
by *metis*
qed
then obtain i **where** $i \geq 0 \wedge i < length\ xs \wedge xs\ !\ i = Unique\ j\ a \wedge (\forall k.\ 0 \leq k \wedge k < i \longrightarrow \neg\ is\text{-}Unique\text{-}i\ j\ (xs\ !\ k))$
using *Cons.hyps*[*of uargs*] *Cons.prems*(*1*) *Cons.prems*(*2*) **by** *auto*
moreover have $\bigwedge k.\ 0 \leq k \wedge k < i + 1 \longrightarrow \neg\ is\text{-}Unique\text{-}i\ j\ ((x\ \#\ xs)\ !\ k)$
proof
fix k **assume** $0 \leq k \wedge k < i + 1$
then show $\neg\ is\text{-}Unique\text{-}i\ j\ ((x\ \#\ xs)\ !\ k)$
apply (*cases k*)
apply (*simp add*: *Shared is-Unique-def is-Unique-i-def*)
by (*simp add*: *calculation*(*2*))
qed
ultimately show *?thesis*
by (*metis Suc-eq-plus1 Suc-less-eq bot-nat-0.extremum length-Cons nth-Cons-Suc*)
next
case (*Unique k uarg*)
then show *?thesis*
proof (*cases* $j = k$)
case *True*
then have $uargs\ j = map\ get\text{-}u\ (filter\ (is\text{-}Unique\text{-}i\ j)\ (x\ \#\ xs))$
by (*meson Cons.prems*(*3*) *possible-sequence-def*)
then have $uarg = a$
by (*simp add*: *True Unique Cons.prems*(*1*) *is-Unique-def is-Unique-i-def*)
then show *?thesis*
using *True Unique* **by** *fastforce*
next
case *False*
moreover have *possible-sequence sargs* (*uargs*(k := *tl* (*uargs k*))) *xs*
proof (*rule possible-sequenceI*)
show $sargs = image\text{-}mset\ get\text{-}s\ (filter\text{-}mset\ is\text{-}Shared\ (mset\ xs))$
by (*metis* (*mono-tags*, *lifting*) *Cons.prems*(*3*) *Unique action.disc*(*2*) *filter-mset-add-mset mset.simps*(*2*) *possible-sequence-def*)
fix i **show** $(uargs(k := tl\ (uargs\ k)))\ i = map\ get\text{-}u\ (filter\ (is\text{-}Unique\text{-}i\ i)\ xs)$
proof (*cases* $i = k$)
case *True*
then show *?thesis*
using *Cons.prems*(*3*) *Unique action.disc*(*2*) *action.sel*(*2*) *filter.simps*(*2*) *fun-upd-same*
is-Unique-def is-Unique-i-def list.sel(*3*) *map-tl possible-sequence-def*[*of sargs uargs x # xs*]
by *metis*
next

```
          case False
          then show ?thesis
             using Cons.prems(3) Unique action.sel(2) filter-remove fun-upd-apply
is-Unique-i-def
              le-numeral-extra(3) length-greater-0-conv list.discI nth-Cons-0 possi-
ble-sequence-def[of sargs uargs x # xs]
            remove-at-index.simps(2) by metis
        qed
      qed
      then obtain i where i ≥ 0 ∧ i < length xs ∧ xs ! i = Unique j a ∧ (∀ k. 0
≤ k ∧ k < i ⟶ ¬ is-Unique-i j (xs ! k))
      by (metis Cons.hyps Cons.prems(1) Cons.prems(2) calculation fun-upd-other)
      moreover have ⋀k. 0 ≤ k ∧ k < i + 1 ⟶ ¬ is-Unique-i j ((x # xs) ! k)
      proof
        fix k assume 0 ≤ k ∧ k < i + 1
        then show ¬ is-Unique-i j ((x # xs) ! k)
          apply (cases k)
           apply (metis False Unique action.sel(2) is-Unique-i-def nth-Cons-0)
          by (simp add: calculation(2))
      qed
      ultimately show ?thesis
      by (metis Suc-eq-plus1 Suc-less-eq bot-nat-0.extremum length-Cons nth-Cons-Suc)
    qed
  qed
qed

lemma remove-at-index-filter:
  assumes i ≥ 0 ∧ i < length s ∧ P (s ! i)
      and ⋀j. j ≥ 0 ∧ j < i ⟹ ¬ P (s ! j)
    shows tl (map get-u (filter P s)) = map get-u (filter P (remove-at-index i s))
  using assms
proof (induct s arbitrary: i)
  case (Cons a s)
  then show ?case
  proof (cases i)
    case 0
    then show ?thesis
      using Cons.prems(1) by auto
  next
    case (Suc k)
    then have tl (map get-u (filter P s)) = map get-u (filter P (remove-at-index k
s))
      apply (cases s)
       apply simp
     by (metis Cons.hyps Cons.prems(1) Cons.prems(2) Suc-less-eq bot-nat-0.extremum
length-Cons nth-Cons-Suc)
    then show ?thesis
      by (metis Cons.prems(2) Suc bot-nat-0.extremum filter.simps(2) nth-Cons-0
remove-at-index.simps(3) zero-less-Suc)
```

qed
qed (*simp*)

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* ≠ []
and *possible-sequence sargs uargs s*
shows ∃ *i*. *i* ≥ *0* ∧ *i* < *length s* ∧ *s* ! *i* = *Unique k a* ∧ *possible-sequence sargs* (*tail-kth uargs k*) (*remove-at-index i s*)
∧ (∀ *j*. *j* ≥ *0* ∧ *j* < *i* ⟶ ¬ *is-Unique-i k* (*s* ! *j*))
proof −
obtain *i* **where** *i* ≥ *0* ∧ *i* < *length s* ∧ *s* ! *i* = *Unique k a* ∧ (∀ *j*. *j* ≥ *0* ∧ *j* < *i* ⟶ ¬ *is-Unique-i k* (*s* ! *j*))
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *head-possible-exists-first-unique*)
let *?s* = *remove-at-index i s*
have *sargs* = *image-mset get-s* (*filter-mset is-Shared* (*mset ?s*))
by (*metis* ‹*0* ≤ *i* ∧ *i* < *length s* ∧ *s* ! *i* = *Unique k a* ∧ (∀ *j*. *0* ≤ *j* ∧ *j* < *i* ⟶ ¬ *is-Unique-i k* (*s* ! *j*))› *action.disc*(*2*) *add.right-neutral assms*(*3*) *filter-single-mset filter-union-mset mset-remove-index possible-sequence-def*)
moreover have *tl* (*uargs k*) = *map get-u* (*filter* (*is-Unique-i k*) *?s*)
proof −
have *uargs k* = *map get-u* (*filter* (*is-Unique-i k*) *s*)
by (*meson assms*(*3*) *possible-sequence-def*)
then show *?thesis*
by (*metis* ‹*0* ≤ *i* ∧ *i* < *length s* ∧ *s* ! *i* = *Unique k a* ∧ (∀ *j*. *0* ≤ *j* ∧ *j* < *i* ⟶ ¬ *is-Unique-i k* (*s* ! *j*))› *action.disc*(*2*) *action.sel*(*2*) *is-Unique-def is-Unique-i-def remove-at-index-filter*)
qed
moreover have *possible-sequence sargs* (*tail-kth uargs k*) (*remove-at-index i s*)
proof (*rule possible-sequenceI*)
show *sargs* = *image-mset get-s* (*filter-mset is-Shared* (*mset* (*remove-at-index i s*)))
by (*simp add*: *calculation*(*1*))
fix *ia* **show** *tail-kth uargs k ia* = *map get-u* (*filter* (*is-Unique-i ia*) (*remove-at-index i s*))
by (*metis* (*mono-tags*, *lifting*) ‹*0* ≤ *i* ∧ *i* < *length s* ∧ *s* ! *i* = *Unique k a* ∧ (∀ *j*. *0* ≤ *j* ∧ *j* < *i* ⟶ ¬ *is-Unique-i k* (*s* ! *j*))› *action.sel*(*2*) *assms*(*3*) *calculation*(*2*) *filter-remove fun-upd-other fun-upd-same is-Unique-i-def possible-sequence-def tail-kth-def*)
qed
ultimately show *?thesis*
using ‹*0* ≤ *i* ∧ *i* < *length s* ∧ *s* ! *i* = *Unique k a* ∧ (∀ *j*. *0* ≤ *j* ∧ *j* < *i* ⟶ ¬ *is-Unique-i k* (*s* ! *j*))› **by** *blast*
qed

lemma *possible-sequence-where-is-unique*:
assumes *possible-sequence sargs uargs* (*Unique k a* # *s*)

shows *a* = *hd* (*uargs k*)
proof −
let *?s* = *Unique k a* # *s*
have *Unique k a* = *hd* (*filter is-Unique ?s*)
by (*simp add*: *is-Unique-def*)
then have *a* = *hd* (*map get-u* (*filter is-Unique ?s*))
by (*simp add*: *is-Unique-def*)
then show *?thesis*
using *action.disc*(*2*) *action.sel*(*2*) *assms filter.simps*(*2*) *hd-map is-Unique-def is-Unique-i-def list.discI list.sel*(*1*) *possible-sequence-def*[*of sargs uargs Unique k a* # *s*]
by *metis*
qed

lemma *possible-sequence-where-is-shared*:
assumes *possible-sequence sargs uargs* (*Shared a* # *s*)
shows *a* ∈# *sargs*
proof −
let *?s* = *Shared a* # *s*
have *a* ∈ *set* (*map get-s* (*filter is-Shared ?s*))
by *simp*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *assms mset-filter mset-map possible-sequence-def set-mset-mset*)
qed

lemma *PRE-unique-tlI*:
assumes *PRE-unique upre qa qb*
and *upre ta tb*
shows *PRE-unique upre* (*ta* # *qa*) (*tb* # *qb*)
proof (*rule PRE-uniqueI*)
show *length* (*ta* # *qa*) = *length* (*tb* # *qb*)
using *PRE-unique-def assms*(*1*) **by** *auto*
fix *i*
show $0 \leq i \wedge i <$ *length* (*tb* # *qb*) ⟹ *upre* ((*ta* # *qa*) ! *i*) ((*tb* # *qb*) ! *i*)
proof (*cases i*)
case *0*
then show *?thesis*
using *assms*(*2*) **by** *auto*
next
case (*Suc k*)
assume $0 \leq i \wedge i <$ *length* (*tb* # *qb*)
then have (*ta* # *qa*) ! *i* = *qa* ! *k* ∧ (*tb* # *qb*) ! *i* = *qb* ! *k*
by (*simp add*: *Suc*)
then show *?thesis* **using** *assms*(*1*) *PRE-unique-def*
using *Suc* ‹$0 \leq i \wedge i <$ *length* (*tb* # *qb*)› **by** *auto*
qed
qed

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*)
using *PRE-sequence-def assms*(*1*) *assms*(*2*) **by** *blast*

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'*
by (*simp add*: *PRE-sequence-def assms*(*1*) *assms*(*2*))

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'*)
using *PRE-sequence-def*[*of spre upre a* # *s b* # *s'*] *PRE-sequence-def*[*of spre upre s s'*]
assms(*1*) *assms*(*2*) *less-Suc-eq-0-disj length-Cons less-Suc-eq-le nth-Cons-0 nth-Cons-Suc*
by *force*

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*
using *PRE-sequence-def*[*of spre upre a* # *s b* # *s'*] *PRE-sequence-def*[*of spre upre s s'*]
apply (*metis Suc-less-eq assms bot-nat-0.extremum diff-Suc-1 length-Cons nth-Cons-Suc*)
by (*metis PRE-sequenceE assms length-Cons list.size*(*3*) *nth-Cons-0 remdups-adj.simps*(*1*) *remdups-adj-length zero-less-Suc*)

fun *compute* :: $('v \Rightarrow 'a \Rightarrow 'v) \Rightarrow ('i \Rightarrow 'v \Rightarrow 'b \Rightarrow 'v) \Rightarrow 'v \Rightarrow ('i, 'a, 'b)$ *action list* $\Rightarrow 'v$ **where**
compute sact uact v0 [] = *v0*
| *compute sact uact v0* (*Shared sarg* # *s*) = *sact* (*compute sact uact v0 s*) *sarg*
| *compute sact uact v0* (*Unique k uarg* # *s*) = *uact k* (*compute sact uact v0 s*) *uarg*

lemma *obtain-other-elem-ms*:
assumes *PRE-shared spre sargs sargs'*
and *sarg* ∈# *sargs*
shows ∃ *sarg'*. *sarg'* ∈# *sargs'* ∧ *spre sarg sarg'* ∧ *PRE-shared spre* (*sargs* − {# *sarg* #}) (*sargs'* − {# *sarg'* #})
proof −
obtain *ms* **where** *asm*: *image-mset fst ms* = *sargs* ∧ *image-mset snd ms* = *sargs'* ∧ (∀ *x* ∈# *ms*. *spre* (*fst x*) (*snd x*))
using *PRE-shared-def assms*(*1*) **by** *blast*
then obtain *x* **where** *x* ∈# *ms fst x* = *sarg*
using *assms*(*2*) **by** *auto*
then have *snd x* ∈# *sargs'* ∧ *spre sarg* (*snd x*)
using *asm* **by** *force*
moreover have *PRE-shared spre* (*sargs* − {# *sarg* #}) (*sargs'* − {# *snd x* #})
proof −
let *?ms* = *ms* − {# *x* #}
have *image-mset fst ?ms* = (*sargs* − {# *sarg* #}) ∧ *image-mset snd ?ms* = (*sargs'* − {# *snd x* #})
by (*simp add*: ‹*fst x* = *sarg*› ‹*x* ∈# *ms*› *asm image-mset-Diff*)
moreover have ⋀*y*. *y* ∈# *?ms* ⟹ *spre* (*fst y*) (*snd y*)
by (*meson asm in-diffD*)
ultimately show *?thesis*
using *PRE-shared-def* **by** *blast*
qed
ultimately show *?thesis*
by *blast*
qed

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''*
using *assms*
proof (*induct s arbitrary*: *sargs uargs sargs' uargs' s'*)
case *Nil*
then have *sargs* = *mset* [] ∧ (∀ *k*. *uargs k* = [])
by (*simp add*: *possible-sequence-def*)
then have *sargs'* = {#} ∧ (∀ *k*. *uargs' k* = [])
by (*metis Nil.prems*(*3*) *Nil.prems*(*4*) *PRE-shared-same-size PRE-unique-def length-0-conv mset.simps*(*1*) *size-eq-0-iff-empty*)
then show ∃ *s''*. *possible-sequence sargs' uargs' s''* ∧ *PRE-sequence spre upre* [] *s''*
by (*simp add*: *PRE-sequence-def possible-sequence-def*)

next
case (*Cons act s*)

show $\exists s''$. *possible-sequence sargs' uargs' s''* $\wedge$ *PRE-sequence spre upre* (*act* # *s*) *s''*
proof (*cases act*)
case (*Shared sarg*)

then have *sarg* $\in$# *sargs*
by (*metis Cons.prems*(*1*) *possible-sequence-where-is-shared*)
then obtain *sarg'* **where** *sarg'* $\in$# *sargs' spre sarg sarg' PRE-shared spre* (*sargs* $-$ {# *sarg* #}) (*sargs'* $-$ {# *sarg'* #})
by (*metis Cons.prems*(*3*) *obtain-other-elem-ms*)
let *?sargs* = *sargs* $-$ {# *sarg* #}
let *?sargs'* = *sargs'* $-$ {# *sarg'* #}
have *possible-sequence ?sargs uargs s*
proof (*rule possible-sequenceI*)
show $\bigwedge i$. *uargs i* = *map get-u* (*filter* (*is-Unique-i i*) *s*)
using *Cons.prems*(*1*) *Shared action.disc*(*1*) *filter.simps*(*2*) *is-Unique-def is-Unique-i-def*
possible-sequence-def[*of sargs uargs act* # *s*]
by *metis*
have *sargs* = *image-mset get-s* (*filter-mset is-Shared* (*mset*(*Shared sarg* # *s*)))
using *Cons.prems*(*1*) *Shared possible-sequence-def* **by** *blast*
then show *sargs* $-$ {#*sarg*#} = *image-mset get-s* (*filter-mset is-Shared* (*mset s*)) **by** *simp*
qed

obtain *i* **where** $i \geq 0 \wedge i <$ *length s'* $\wedge$ *s'* ! *i* = *Shared sarg'* $\wedge$ *possible-sequence ?sargs' uargs'* (*remove-at-index i s'*)
by (*meson Cons.prems*(*2*) ‹*sarg'* $\in$# *sargs'*› *exists-index-in-sequence-shared*)

then obtain *s''* **where** *possible-sequence ?sargs' uargs' s''* $\wedge$ *PRE-sequence spre upre s s''*
using *Cons.hyps Cons.prems*(*4*) ‹*PRE-shared spre* (*sargs* $-$ {#*sarg*#}) (*sargs'* $-$ {#*sarg'*#})› ‹*possible-sequence* (*sargs* $-$ {#*sarg*#}) *uargs s*› **by** *blast*

let *?s''* = *Shared sarg'* # *s''*

have *possible-sequence sargs' uargs' ?s''*
proof (*rule possible-sequenceI*)
show $\bigwedge i$. *uargs' i* = *map get-u* (*filter* (*is-Unique-i i*) (*Shared sarg'* # *s''*))
by (*metis* ‹*possible-sequence* (*sargs'* $-$ {#*sarg'*#}) *uargs' s''* $\wedge$ *PRE-sequence spre upre s s''*› *action.disc*(*1*) *filter.simps*(*2*) *is-Unique-def is-Unique-i-def possible-sequence-def*)
show *sargs'* = *image-mset get-s* (*filter-mset is-Shared* (*mset* (*Shared sarg'* # *s''*)))
using ‹*possible-sequence* (*sargs'* $-$ {#*sarg'*#}) *uargs' s''* $\wedge$ *PRE-sequence spre upre s s''*› ‹*sarg'* $\in$# *sargs'*›

insert-DiffM[*of sarg′ sargs′*] *possible-sequence-def*[*of sargs′* − {#*sarg′*#} *uargs′ s″*]
action.disc(*1*) *action.sel*(*1*) *filter-mset-add-mset msed-map-invL mset.simps*(*2*)
by *auto*
qed

moreover have *PRE-sequence spre upre* (*act* # *s*) *?s″*
by (*simp add*: *PRE-sequenceI-rec Shared* ‹*possible-sequence* (*sargs′* − {#*sarg′*#}) *uargs′ s″* ∧ *PRE-sequence spre upre s s″*› ‹*spre sarg sarg′*›)

ultimately show ∃ *s″*. *possible-sequence sargs′ uargs′ s″* ∧ *PRE-sequence spre upre* (*act* # *s*) *s″* **by** *auto*
next
case (*Unique k uarg*)
then have *hd* (*uargs k*) = *uarg*
by (*metis Cons.prems*(*1*) *possible-sequence-where-is-unique*)
moreover have *uargs k* ≠ []
by (*metis* (*no-types*, *lifting*) *Cons.prems*(*1*) *Unique action.disc*(*2*) *action.sel*(*2*) *dropWhile-eq-Cons-conv dropWhile-eq-self-iff filter.simps*(*2*) *is-Unique-def is-Unique-i-def list.map-disc-iff possible-sequence-def*)
ultimately have *uargs k* = *uarg* # *tl* (*uargs k*)
by *fastforce*
moreover have *uargs′ k* = *hd* (*uargs′ k*) # *tl* (*uargs′ k*)
by (*metis Cons.prems*(*4*) *PRE-unique-def calculation length-Cons list.exhaust-sel list.size*(*3*) *nat.simps*(*3*))
ultimately have *upre k uarg* (*hd* (*uargs′ k*))
by (*metis Cons.prems*(*4*) *PRE-unique-def length-greater-0-conv list.simps*(*3*) *list.size*(*3*) *nth-Cons-0 remdups-adj.simps*(*1*) *remdups-adj-length*)
moreover have *PRE-unique* (*upre k*) (*tl* (*uargs k*)) (*tl* (*uargs′ k*))
by (*metis Cons.prems*(*4*) *PRE-unique-implies-tl* ‹*uargs k* = *uarg* # *tl* (*uargs k*)› ‹*uargs′ k* = *hd* (*uargs′ k*) # *tl* (*uargs′ k*)›)
moreover have *possible-sequence sargs* (*tail-kth uargs k*) *s*
proof (*rule possible-sequenceI*)
show ⋀*i*. *tail-kth uargs k i* = *map get-u* (*filter* (*is-Unique-i i*) *s*)
proof −
fix *i*
obtain *ii* **where** *asms*: *ii* ≥ *0 ii* < *length* (*act* # *s*) ∧
(*act* # *s*) ! *ii* = *Unique k uarg* ∧
possible-sequence sargs (*tail-kth uargs k*) (*remove-at-index ii* (*act* # *s*)) ∧ (∀ *j*. *0* ≤ *j* ∧ *j* < *ii* ⟶ ¬ *is-Unique-i k* ((*act* # *s*) ! *j*))
by (*metis Cons.prems*(*1*) ‹*hd* (*uargs k*) = *uarg*› ‹*uargs k* ≠ []› *exists-index-in-sequence-unique*)
then show *tail-kth uargs k i* = *map get-u* (*filter* (*is-Unique-i i*) *s*)
by (*metis Unique action.disc*(*2*) *action.sel*(*2*) *bot-nat-0.extremum bot-nat-0.not-eq-extremum is-Unique-def is-Unique-i-def nth-Cons-0 possible-sequence-def remove-at-index.simps*(*2*))
qed
show *sargs* = *image-mset get-s* (*filter-mset is-Shared* (*mset s*))
by (*metis* (*mono-tags*, *lifting*) *Cons.prems*(*1*) *Unique action.disc*(*2*) *filter-mset-add-mset mset.simps*(*2*) *possible-sequence-def*)

qed
let *?uarg′ = hd (uargs′ k)*

obtain *i* **where** $i \geq 0 \wedge i < length\ s' \wedge s' \ !\ i = Unique\ k\ ?uarg' \wedge$ *possible-sequence sargs′ (tail-kth uargs′ k) (remove-at-index i s′)*
by *(metis Cons.prems(2) ‹uargs′ k = hd (uargs′ k) # tl (uargs′ k)› exists-index-in-sequence-unique list.discI)*
then obtain *s′′* **where** *possible-sequence sargs′ (tail-kth uargs′ k) s′′ ∧ PRE-sequence spre upre s s′′*
using *Cons.hyps[of sargs tail-kth uargs k sargs′ tail-kth uargs′ k] Cons.prems(3) ‹possible-sequence sargs (tail-kth uargs k) s› calculation(2)*
Cons.prems(4) fun-upd-other fun-upd-same tail-kth-def
by *metis*

let *?s′′ = Unique k (hd (uargs′ k)) # s′′*
have *PRE-sequence spre upre (act # s) ?s′′*
by *(simp add: PRE-sequenceI-rec Unique ‹possible-sequence sargs′ (tail-kth uargs′ k) s′′ ∧ PRE-sequence spre upre s s′′› calculation(1))*
moreover have *possible-sequence sargs′ uargs′ ?s′′*
proof *(rule possible-sequenceI)*
show $\bigwedge i.$ *uargs′ i = map get-u (filter (is-Unique-i i) (Unique k (hd (uargs′ k)) # s′′))*
proof −
fix *i*
obtain *ii* **where** *ii-def*: $ii \geq 0 \wedge ii < length\ s' \wedge$
$s' \ !\ ii = Unique\ k\ (hd\ (uargs'\ k)) \wedge$
possible-sequence sargs′ (tail-kth uargs′ k) (remove-at-index ii s′) $\wedge (\forall j.\ 0 \leq j \wedge j < ii \longrightarrow \neg\ is\text{-}Unique\text{-}i\ k\ (s' \ !\ j))$
by *(metis Cons.prems(2) ‹uargs′ k = hd (uargs′ k) # tl (uargs′ k)› exists-index-in-sequence-unique list.discI)*
then show *uargs′ i = map get-u (filter (is-Unique-i i) (Unique k (hd (uargs′ k)) # s′′))*
using *filter-remove[of ii s′ is-Unique-i i] remove-at-index-filter[of ii s′ is-Unique-i i]*
Cons.prems(2) ‹possible-sequence sargs′ (tail-kth uargs′ k) s′′ ∧ PRE-sequence spre upre s s′′›
‹uargs′ k = hd (uargs′ k) # tl (uargs′ k)› action.sel(2) action.sel(3)
filter.simps(2)[of is-Unique-i i Unique k (hd (uargs′ k)) s′′]
is-Unique-i-def list.simps(9)[of get-u]
possible-sequence-def[of sargs′ tail-kth uargs′ k remove-at-index ii s′]
possible-sequence-def[of sargs′ uargs′ s′]
possible-sequence-def[of sargs′ tail-kth uargs′ k s′′] ii-def
by *metis*
qed
show *sargs′ = image-mset get-s (filter-mset is-Shared (mset (Unique k (hd (uargs′ k)) # s′′)))*
using *‹possible-sequence sargs′ (tail-kth uargs′ k) s′′ ∧ PRE-sequence spre upre s s′′› possible-sequence-def* **by** *auto*
qed

ultimately show $\exists s''$. *possible-sequence sargs' uargs' s''* $\wedge$ *PRE-sequence spre upre* (*act* # *s*) *s''* **by** *blast*
qed
qed

lemma *insert-remove-same-list*:
assumes $k \geq 0 \wedge k < length\ s$
and $s\ !\ k = x$
shows $s = insert\text{-}at\ k\ x\ (remove\text{-}at\text{-}index\ k\ s)$
proof (*rule list-ext*)
show $length\ s = length\ (insert\text{-}at\ k\ x\ (remove\text{-}at\text{-}index\ k\ s))$
by (*metis One-nat-def Suc-pred add.commute assms*(*1*) *insert-at-index*(*1*) *length-greater-0-conv less-Suc-eq-le linorder-not-le list.size*(*3*) *plus-1-eq-Suc remove-at-index*(*1*))
fix *i* **assume** *asm0*: $0 \leq i \wedge i < length\ s$
show $s\ !\ i = insert\text{-}at\ k\ x\ (remove\text{-}at\text{-}index\ k\ s)\ !\ i$
proof (*cases* $i < k$)
case *True*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-pred asm0 assms*(*1*) *insert-at-index*(*2*) *less-Suc-eq-le order-le-less-trans remove-at-index*(*1*) *remove-at-index*(*2*))
next
case *False*
then have $i \geq k$ **by** *simp*
then show *?thesis*
proof (*cases* $i = k$)
case *True*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-pred assms*(*1*) *assms*(*2*) *insert-at-index*(*3*) *less-Suc-eq-le order-le-less-trans remove-at-index*(*1*))
next
case *False*
then have $i > k$
using ‹$k \leq i$› *nless-le* **by** *blast*
then show $s\ !\ i = insert\text{-}at\ k\ x\ (remove\text{-}at\text{-}index\ k\ s)\ !\ i$
apply (*cases i*)
apply *blast*
using *Groups.add-ac*(*2*) *One-nat-def Suc-less-eq Suc-pred asm0 assms*(*1*)
insert-at-index(*4*)[*of k - i x*]
less-Suc-eq-le order-le-less-trans plus-1-eq-Suc remove-at-index(*1*)[*of k s*]
remove-at-index(*3*)[*of k s*]
by *fastforce*
qed
qed
qed

lemma *swap-works*:
assumes $length\ s = length\ s'$
and $k < length\ s - 1$
and $\bigwedge i.\ i \geq 0 \wedge i < length\ s \wedge i \neq k \wedge i \neq k + 1 \Longrightarrow s\ !\ i = s'\ !\ i$

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 ¬ (∃ *k′*. *is-Unique-i k′* (*s* ! *k*) ∧ *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*)
using *assms*
proof (*induct k arbitrary*: *s s′*)
case *0*
then obtain *x1 x2 xs* **where** *s* = *x1* # *x2* # *xs*
by (*metis Suc-length-conv Suc-pred add-0 le-add-diff-inverse less-diff-conv less-imp-le-nat plus-1-eq-Suc*)
then have *hd s′* = *x2*
by (*metis 0.prems*(*1*) *0.prems*(*2*) *0.prems*(*5*) *One-nat-def add-0 hd-conv-nth length-greater-0-conv length-tl list.sel*(*2*) *nth-Cons-0 nth-Cons-Suc*)
moreover have *hd* (*tl s′*) = *x1*
by (*metis 0.prems*(*1*) *0.prems*(*2*) *0.prems*(*4*) *Suc-eq-plus1* ‹*s* = *x1* # *x2* # *xs*› *hd-conv-nth length-greater-0-conv length-tl nth-Cons-0 nth-tl*)
ultimately obtain *xs′* **where** *s′* = *x2* # *x1* # *xs′*
by (*metis 0.prems*(*1*) *0.prems*(*2*) *length-greater-0-conv length-tl list.collapse list.sel*(*2*))
moreover have *xs* = *xs′*
proof (*rule list-ext*)
show *length xs* = *length xs′*
using *0.prems*(*1*) ‹*s* = *x1* # *x2* # *xs*› *calculation* **by** *auto*
fix *i* **assume** *0* ≤ *i* ∧ *i* < *length xs*
then show *xs* ! *i* = *xs′* ! *i*
by (*metis 0.prems*(*3*) *Suc-eq-plus1 Suc-less-eq* ‹*s* = *x1* # *x2* # *xs*› *bot-nat-0.extremum calculation diff-Suc-1 length-Cons nat.simps*(*3*) *nth-Cons-Suc*)
qed
have *PRE-sequence spre upre xs xs*
apply (*cases x1*) **apply** (*cases x2*)
using *0.prems*(*6*) ‹*s* = *x1* # *x2* # *xs*› *PRE-sequenceE-rec*(*1*) **by** *blast+*
then have *α* (*compute sact uact v0 xs*) = *α* (*compute sact uact v0′ xs*)
using *assms*(*7*)
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then have *α* (*compute sact uact v0 xs*) = *α* (*compute sact uact v0′ xs*)
using *PRE-sequenceE-rec*(*1*) **by** *blast*
then show *α* (*compute sact uact v0* (*a* # *xs*)) = *α* (*compute sact uact v0′* (*a* # *xs*))
proof (*cases a*)
case (*Shared x1*)
then show *?thesis*
by (*metis* ‹*all-axioms α sact spre uact upre*› *Cons.prems*(*1*) *PRE-sequenceE-rec*(*2*)

‹α (*compute sact uact v0 xs*) = α (*compute sact uact v0' xs*)› *abstract-pre.simps*(*1*) *compute.simps*(*2*) *sabstract*)
next
case (*Unique x2*)
then show *?thesis*
by (*metis* ‹*all-axioms* α *sact spre uact upre*› *Cons.prems*(*1*) *PRE-sequenceE-rec*(*2*) ‹α (*compute sact uact v0 xs*) = α (*compute sact uact v0' xs*)› *abstract-pre.simps*(*2*) *compute.simps*(*3*) *uabstract*)
qed
qed
then show *?case*
proof (*cases x1*)
case (*Shared sarg1*)
then have *x1* = *Shared sarg1* **by** *simp*
then show *?thesis*
proof (*cases x2*)
case (*Shared sarg2*)
then show α (*compute sact uact v0 s*) = α (*compute sact uact v0' s'*)
using ‹*all-axioms* α *sact spre uact upre*› *0.prems*(*2*) *0.prems*(*5*) *0.prems*(*6*) *One-nat-def*
PRE-sequenceE-rec(*2*)[*of spre upre x1 x2 # xs x1 x2 # xs*]
PRE-sequence-def[*of spre upre s s*] *Suc-eq-plus1*
‹α (*compute sact uact v0 xs*) = α (*compute sact uact v0' xs*)› ‹*s* = *x1 # x2 # xs*›
‹*x1* = *Shared sarg1*› ‹*xs* = *xs'*›
abstract-pre.simps(*1*)[*of spre upre sarg2 sarg2*]
abstract-pre.simps(*1*)[*of spre upre sarg1 sarg1*]
calculation compute.simps(*2*)[*of sact uact v0*]
calculation compute.simps(*2*)[*of sact uact v0'*]
nth-Cons-0 ss-com[*of* α *sact spre uact upre*] *zero-less-diff zero-less-one-class.zero-le-one*
by *metis*
next
case (*Unique uarg2*)
then show *?thesis*
using ‹*all-axioms* α *sact spre uact upre*› *0.prems*(*6*) *PRE-sequenceE-rec*(*1*)[*of spre upre x1 x2 # xs x1 x2 # xs*]
PRE-sequenceE-rec(*2*)[*of spre upre*]
Shared ‹α (*compute sact uact v0 xs*) = α (*compute sact uact v0' xs*)› ‹*s* = *x1 # x2 # xs*› ‹*xs* = *xs'*›
abstract-pre.simps(*1*)[*of spre upre*] *abstract-pre.simps*(*2*)[*of spre upre*] *calculation*
compute.simps(*2*)[*of sact uact*] *compute.simps*(*3*)[*of sact uact*]
su-com[*of* α *sact spre uact upre*]
by *metis*
qed
next
case (*Unique k1 uarg1*)
then have *x1* = *Unique k1 uarg1* **by** *simp*
then show *?thesis*

proof (*cases x2*)
case (*Shared sarg2*)
then have *spre sarg2 sarg2* $\wedge$ *upre k1 uarg1 uarg1*
by (*metis 0.prems*(*6*) *PRE-sequenceE-rec*(*1*) *PRE-sequenceE-rec*(*2*) *Unique* ‹*s* = *x1* # *x2* # *xs*› *abstract-pre.simps*(*1*) *abstract-pre.simps*(*2*))
then show *?thesis*
using ‹*all-axioms* α *sact spre uact upre*› *Unique* ‹α (*compute sact uact v0 xs*) = α (*compute sact uact v0′ xs*)›
‹*s* = *x1* # *x2* # *xs*› ‹*s′* = *x2* # *x1* # *xs′*› ‹*xs* = *xs′*› *compute.simps*(*2*)[*of sact uact*]
compute.simps(*3*)[*of sact uact*] *su-com*[*of* α *sact spre uact upre*]
by (*metis Shared*)
next
case (*Unique k2 uarg2*)
then have *k1* $\neq$ *k2*
by (*metis 0.prems*(*5*) *0.prems*(*8*) *Suc-eq-plus1* ‹$\bigwedge$*thesis.* ($\bigwedge$*xs′. s′* = *x2* # *x1* # *xs′* $\Longrightarrow$ *thesis*) $\Longrightarrow$ *thesis*› ‹*s* = *x1* # *x2* # *xs*› ‹*x1* = *Unique k1 uarg1*› *action.disc*(*2*) *action.sel*(*2*) *is-Unique-def is-Unique-i-def nth-Cons-0*)
then have *upre k2 uarg2 uarg2* $\wedge$ *upre k1 uarg1 uarg1*
by (*metis 0.prems*(*6*) *PRE-sequenceE-rec*(*1*) *PRE-sequenceE-rec*(*2*) *Unique* ‹*s* = *x1* # *x2* # *xs*› ‹*x1* = *Unique k1 uarg1*› *abstract-pre.simps*(*2*))

then show *?thesis*
using ‹*all-axioms* α *sact spre uact upre*› *Unique* ‹α (*compute sact uact v0 xs*) = α (*compute sact uact v0′ xs*)›
‹*s* = *x1* # *x2* # *xs*› ‹*s′* = *x2* # *x1* # *xs′*› ‹*xs* = *xs′*› *compute.simps*(*2*)[*of sact uact*]
uu-com[*of* α *sact spre uact upre k1 k2 compute sact uact v0′ xs compute sact uact v0 xs*]
‹*k1* $\neq$ *k2*› ‹*x1* = *Unique k1 uarg1*› *compute.simps*(*3*)
by *auto*
qed
qed
next
case (*Suc k*)
then obtain *x xs x′ xs′* **where** *s* = *x* # *xs s′* = *x′* # *xs′*
by (*metis diff-0-eq-0 length-0-conv neq-Nil-conv not-less-zero*)
then have *x* = *x′*
using *Suc.prems*(*3*) **by** *force*
moreover have α (*compute sact uact v0* (*tl s*)) = α (*compute sact uact v0′* (*tl s′*))
proof (*rule Suc*(*1*))
show *length* (*tl s*) = *length* (*tl s′*)
by (*simp add*: *Suc.prems*(*1*))
show *k* < *length* (*tl s*) − *1*
using *Suc.prems*(*2*) **by** *auto*
show $\bigwedge$*i. 0* $\leq$ *i* $\wedge$ *i* < *length* (*tl s*) $\wedge$ *i* $\neq$ *k* $\wedge$ *i* $\neq$ *k* + *1* $\Longrightarrow$ *tl s* ! *i* = *tl s′* ! *i*
by (*metis Suc.prems*(*3*) *Suc-eq-plus1* ‹*length* (*tl s*) = *length* (*tl s′*)› *length-tl less-diff-conv nat.inject nat-le-linear not-less-eq-eq nth-tl*)

show *tl s ! k = tl s′ ! (k + 1)*
by (*metis Suc.prems(4) Suc-eq-plus1* ‹*s = x # xs*› ‹*s′ = x′ # xs′*› *add-diff-cancel-right′ add-gr-0 le-neq-implies-less list.sel(3) not-one-le-zero nth-Cons-pos zero-less-one-class.zero-le-one*)
show *tl s′ ! k = tl s ! (k + 1)*
by (*metis Suc.prems(5) Suc-eq-plus1* ‹*s = x # xs*› ‹*s′ = x′ # xs′*› *list.sel(3) nth-Cons-Suc*)
show *PRE-sequence spre upre (tl s) (tl s)*
by (*metis Suc.prems(6)* ‹*s = x # xs*› *PRE-sequenceE-rec(1) list.sel(3)*)
show *α v0 = α v0′*
by (*simp add: assms(7)*)
show ¬ (∃ *k′. is-Unique-i k′ (tl s ! k)* ∧ *is-Unique-i k′ (tl s ! (k + 1))*)
using *Suc.prems(8)* ‹*s = x # xs*› **by** *force*
show *all-axioms α sact spre uact upre*
by (*simp add: Suc.prems(9)*)
qed
ultimately show *?case*
proof (*cases x*)
case (*Shared x1*)
then show *?thesis*
using ‹*all-axioms α sact spre uact upre*› *PRE-sequenceE-rec(2) Suc.prems(6)* ‹*α (compute sact uact v0 (tl s)) = α (compute sact uact v0′ (tl s′))*› ‹*s = x # xs*› ‹*s′ = x′ # xs′*› ‹*x = x′*› *sabstract*
by *fastforce*
next
case (*Unique x2*)
then show *?thesis*
using ‹*all-axioms α sact spre uact upre*› *PRE-sequenceE-rec(2) Suc.prems(6)*
‹*α (compute sact uact v0 (tl s)) = α (compute sact uact v0′ (tl s′))*› ‹*s = x # xs*› ‹*s′ = x′ # xs′*›
‹*x = x′*› *uabstract[of α sact spre uact upre]*
by *fastforce*
qed
qed

lemma *mset-remove*:
assumes *k* ≥ *0* ∧ *k < length s*
shows *mset s = mset (remove-at-index k s) + {# s ! k #}*
using *assms*
proof (*induct s arbitrary: k*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a s*)
then show *?case*
using *less-Suc-eq-0-disj* **by** *auto*
qed

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*
apply (*cases a*)
apply (*cases b*)
using *abstract-pre.simps*(*1*) *assms spre-refl* **apply** *metis*
using *assms* **apply** *force*
apply (*cases b*)
using *assms* **apply** *force*
using *abstract-pre.simps*(*2*) *assms upre-refl* **by** *metis*

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 (*rule PRE-sequenceI*)
show *length* s' = *length* s'
by *simp*
fix *i* **assume** $0 \leq i \wedge i <$ *length* s'
then show *abstract-pre spre upre* (s' ! *i*) (s' ! *i*)
by (*metis PRE-sequence-def abstract-pre-refl assms*)
qed

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*)
using *assms*
proof (*induct n arbitrary*: *s*)
case *0*
then show *?case*
by (*metis PRE-sequenceE-rec*(*1*) *nat.simps*(*3*) *remove-at-index.elims*)
next
case (*Suc n*)
then show *?case*
apply (*cases s*)
apply *force*
by (*metis PRE-sequenceE-rec*(*1*) *PRE-sequenceE-rec*(*2*) *PRE-sequenceI-rec remove-at-index.simps*(*3*))
qed

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*)
using *assms*
proof (*induct n arbitrary*: *s*)
case *0*
then show *?case*
by (*simp add*: *PRE-sequenceI-rec*)

next
  case (*Suc n*)
  then show *?case*
    apply (*cases s*)
     apply (*simp add*: *PRE-sequenceI-rec*)
    by (*metis PRE-sequenceE-rec*(*1*) *PRE-sequenceE-rec*(*2*) *PRE-sequenceI-rec insert-at.simps*(*3*))
qed

lemma *empty-possible-sequence*:
  assumes *possible-sequence sargs uargs* []
      and *possible-sequence sargs uargs* s'
  shows $s' = []$
proof (*rule ccontr*)
  assume $s' \neq []$
  then obtain *x q* **where** $s' = x \# q$
    by (*meson neq-Nil-conv*)
  then show *False*
  proof (*cases x*)
    case (*Shared x1*)
    then show *?thesis*
       by (*metis* ‹$s' = x \# q$› *assms*(*1*) *assms*(*2*) *exists-index-in-sequence-shared less-zeroE list.size*(*3*) *possible-sequence-where-is-shared*)
  next
    case (*Unique k uarg*)
    then have *uargs k* $\neq []$
     by (*metis* (*no-types*, *lifting*) ‹$s' = x \# q$› *action.disc*(*2*) *action.sel*(*2*) *assms*(*2*) *filter.simps*(*2*) *is-Unique-def is-Unique-i-def list.discI list.map-disc-iff possible-sequence-def*)
    then show *?thesis*
     by (*metis assms*(*1*) *exists-index-in-sequence-unique less-nat-zero-code list.size*(*3*))
  qed
qed

lemma *it-all-commutes*:
  assumes *possible-sequence sargs uargs s*
      and *possible-sequence sargs uargs* s'
      and $\alpha\ v0 = \alpha\ v0'$
      and *PRE-sequence spre upre s s*
      and *PRE-sequence spre upre* $s'\ s'$
      and *all-axioms* α *sact spre uact upre*
    shows α (*compute sact uact v0 s*) $= \alpha$ (*compute sact uact* $v0'\ s'$)
  using *assms*
proof (*induct size s arbitrary*: *sargs uargs s* s')
  case *0*
  then have $s = [] \land s' = []$
    by (*simp add*: *empty-possible-sequence*)
  then show *?case*
    by (*simp add*: *0.prems*(*1*) *0.prems*(*2*) *assms*(*3*))
next

case (*Suc n*)
moreover obtain *x s1* **where** $s = x \# s1$
by (*meson Suc.hyps*(*2*) *Suc-length-conv*)
then have *abstract-pre spre upre x x*
using *Suc.prems*(*4*) *PRE-sequenceE-rec*(*2*) **by** *blast*
then show *?case*
proof (*cases x*)
case (*Shared sarg*)
then have *Shared sarg* $\in$ *set* s'
by (*metis Suc.prems*(*1*) *Suc.prems*(*2*) ‹$s = x \# s1$› *exists-index-in-sequence-shared nth-mem possible-sequence-where-is-shared*)
then obtain *k* **where** $k \geq 0 \wedge k < \mathit{length}\ s' \wedge s' \mathbin{!} k = x$
by (*metis Shared bot-nat-0.extremum in-set-conv-nth*)

let $?s' =$ *remove-at-index* $k\ s'$
have *length* $?s' = \mathit{length}\ s' - 1$
by (*simp add*: ‹$0 \leq k \wedge k < \mathit{length}\ s' \wedge s' \mathbin{!} k = x$› *remove-at-index*(*1*))
moreover have $k < \mathit{length}\ s'$
by (*simp add*: ‹$0 \leq k \wedge k < \mathit{length}\ s' \wedge s' \mathbin{!} k = x$›)
then have $s' =$ *insert-at* $k\ x\ ?s'$
by (*simp add*: ‹$0 \leq k \wedge k < \mathit{length}\ s' \wedge s' \mathbin{!} k = x$› *insert-remove-same-list*)
define i :: *nat* **where** $i = k$
have $i \geq 0 \wedge i \leq k \Longrightarrow \alpha$ (*compute sact uact v0'* (*insert-at* $(k - i)\ x\ ?s'$)) $= \alpha$ (*compute sact uact v0'* s')
proof (*induct i*)
case *0*
then show *?case*
using ‹$s' =$ *insert-at* $k\ x$ (*remove-at-index* $k\ s'$)› **by** *auto*
next
case (*Suc i*)
then have α (*compute sact uact v0'* (*insert-at* $(k - i)\ x$ (*remove-at-index* $k\ s'$))) $= \alpha$ (*compute sact uact v0'* s')
using *Suc-leD* **by** *blast*
moreover have α (*compute sact uact v0'* (*insert-at* $(k - \mathit{Suc}\ i)\ x$ (*remove-at-index* $k\ s'$))) $= \alpha$ (*compute sact uact v0'* (*insert-at* $(k - i)\ x$ (*remove-at-index* $k\ s'$)))
proof (*rule swap-works*)
show *length* (*insert-at* $(k - \mathit{Suc}\ i)\ x$ (*remove-at-index* $k\ s'$)) $=$ *length* (*insert-at* $(k - i)\ x$ (*remove-at-index* $k\ s'$))
by (*metis* (*no-types, lifting*) *Suc-pred'* ‹$0 \leq k \wedge k < \mathit{length}\ s' \wedge s' \mathbin{!} k = x$› ‹*length* (*remove-at-index* $k\ s'$) $= \mathit{length}\ s' - 1$› *diff-le-self insert-at-index*(*1*) *less-Suc-eq-le order-le-less-trans*)
show *PRE-sequence spre upre* (*insert-at* $(k - \mathit{Suc}\ i)\ x$ (*remove-at-index* $k\ s'$)) (*insert-at* $(k - \mathit{Suc}\ i)\ x$ (*remove-at-index* $k\ s'$))
proof −
have *PRE-sequence spre upre* (*remove-at-index* $k\ s'$) (*remove-at-index* $k\ s'$) **using** ‹*PRE-sequence spre upre* $s'\ s'$›
using *PRE-sequence-removes* **by** *auto*
then show *?thesis* **using** *PRE-sequence-insert* ‹*abstract-pre spre upre x x*› **by** *blast*

qed
show *α v0′ = α v0′* **by** *simp*
let *?k = k − Suc i*

show *?k < length (insert-at (k − Suc i) x (remove-at-index k s′)) − 1*
using *One-nat-def Suc.prems Suc-diff-Suc Suc-le-lessD ‹k < length s′› ‹length (remove-at-index k s′) = length s′ − 1›*
‹s′ = insert-at k x (remove-at-index k s′)› diff-le-self diff-zero
insert-at-index(1)[of k − Suc i - x] insert-at-index(1)[of k - x] less-Suc-eq-le order-le-less-trans
by *simp*
show *insert-at (k − Suc i) x (remove-at-index k s′) ! ?k = insert-at (k − i) x (remove-at-index k s′) ! (?k + 1)*
apply *(cases k)*
using *Suc.prems* **apply** *blast*
apply *(cases ?k)*
apply *(metis (no-types, lifting) Suc.prems Suc-eq-plus1 Suc-leI ‹k − Suc i < length (insert-at (k − Suc i) x (remove-at-index k s′)) − 1› add-diff-cancel-right′ diff-diff-cancel diff-zero insert-at-index(1) insert-at-index(3) le-numeral-extra(3) length-greater-0-conv list.size(3) nat-less-le plus-1-eq-Suc)*
proof −
fix *nat nata* **assume** *r: k = Suc nat k − Suc i = Suc nata*
moreover have *insert-at (k − i) x (remove-at-index k s′) ! (k − i) = x*
by *(metis Suc-pred′ ‹k < length s′› ‹length (remove-at-index k s′) = length s′ − 1› bot-nat-0.extremum diff-le-self insert-at-index(3) less-Suc-eq-le order-le-less-trans)*
moreover have *⋀x. insert-at (k − Suc i) x (remove-at-index k s′) ! (k − Suc i) = x*
by *(metis Suc-leI Suc-le-mono Suc-pred′ ‹0 ≤ k ∧ k < length s′ ∧ s′ ! k = x› ‹length (remove-at-index k s′) = length s′ − 1› bot-nat-0.extremum diff-le-self insert-at-index(3) order-le-less-trans)*
ultimately show *?thesis*
by *(metis Suc.prems Suc-diff-Suc Suc-eq-plus1 Suc-le-lessD)*
qed
show *insert-at (k − i) x (remove-at-index k s′) ! ?k = insert-at (k − Suc i) x (remove-at-index k s′) ! (?k + 1)*
proof −
have *insert-at (k − i) x (remove-at-index k s′) ! (k − Suc i) = remove-at-index k s′ ! (k − Suc i)*
by *(metis (no-types, lifting) Suc.prems Suc-diff-Suc Suc-eq-plus1 Suc-leI Suc-le-lessD ‹k < length s′› ‹length (remove-at-index k s′) = length s′ − 1› add-leE insert-at-index(2) le-add-diff-inverse2 le-add-same-cancel2 lessI less-Suc-eq-le)*
moreover have *length (insert-at (k − Suc i) x (remove-at-index k s′)) = length (remove-at-index k s′) + 1*
by *(metis Suc-eq-plus1 ‹0 ≤ k ∧ k < length s′ ∧ s′ ! k = x› ‹length (remove-at-index k s′) = length s′ − 1› add-le-imp-le-diff insert-at-index(1) less-eq-Suc-le less-imp-diff-less)*
then have *insert-at (k − Suc i) x (remove-at-index k s′) ! (k − Suc i + 1) = remove-at-index k s′ ! (k − Suc i + 1 − 1)*

by (*metis* ‹$k - Suc\ i < length\ (insert\text{-}at\ (k - Suc\ i)\ x\ (remove\text{-}at\text{-}index\ k\ s')) - 1$› *add-diff-cancel-right′ insert-at-index*(*4*) *less-add-one less-diff-conv less-imp-le-nat*)
ultimately show *?thesis*
by *simp*
qed

show $\neg\ (\exists k'.\ is\text{-}Unique\text{-}i\ k'\ (insert\text{-}at\ (k - Suc\ i)\ x\ (remove\text{-}at\text{-}index\ k\ s')\ !\ (k - Suc\ i)) \wedge is\text{-}Unique\text{-}i\ k'\ (insert\text{-}at\ (k - Suc\ i)\ x\ (remove\text{-}at\text{-}index\ k\ s')\ !\ (k - Suc\ i + 1)))$
by (*metis* (*no-types*, *lifting*) *One-nat-def Shared Suc.prems Suc-diff-Suc* ‹$k < length\ s'$› ‹$length\ (remove\text{-}at\text{-}index\ k\ s') = length\ s' - 1$› *action.disc*(*1*) *add-leE diff-zero insert-at-index*(*3*) *is-Unique-def is-Unique-i-def le-add-diff-inverse2 le-add-same-cancel2 less-Suc-eq-le order-le-less-trans*)

show $\bigwedge j.\ 0 \le j \wedge j < length\ (insert\text{-}at\ (k - Suc\ i)\ x\ (remove\text{-}at\text{-}index\ k\ s')) \wedge j \ne ?k \wedge j \ne ?k + 1 \Longrightarrow insert\text{-}at\ (k - Suc\ i)\ x\ (remove\text{-}at\text{-}index\ k\ s')\ !\ j = insert\text{-}at\ (k - i)\ x\ (remove\text{-}at\text{-}index\ k\ s')\ !\ j$
proof (*clarify*)
fix *j* **assume** $0 \le j\ j < length\ (insert\text{-}at\ (k - Suc\ i)\ x\ (remove\text{-}at\text{-}index\ k\ s'))\ j \ne k - Suc\ i\ j \ne k - Suc\ i + 1$
moreover have $length\ (insert\text{-}at\ (k - Suc\ i)\ x\ (remove\text{-}at\text{-}index\ k\ s')) = length\ (remove\text{-}at\text{-}index\ k\ s') + 1$
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc.prems Suc-diff-Suc* ‹$k < length\ s'$› ‹$length\ (remove\text{-}at\text{-}index\ k\ s') = length\ s' - 1$› *add-leE diff-zero insert-at-index*(*1*) *le-add-diff-inverse2 less-Suc-eq-le order-le-less-trans*)
moreover have $k - Suc\ i \le length\ (remove\text{-}at\text{-}index\ k\ s')$
using ‹$k - Suc\ i < length\ (insert\text{-}at\ (k - Suc\ i)\ x\ (remove\text{-}at\text{-}index\ k\ s')) - 1$› *calculation*(*5*) **by** *force*
ultimately show $insert\text{-}at\ (k - Suc\ i)\ x\ (remove\text{-}at\text{-}index\ k\ s')\ !\ j = insert\text{-}at\ (k - i)\ x\ (remove\text{-}at\text{-}index\ k\ s')\ !\ j$
apply (*cases* $j < k - Suc\ i$)
using *insert-at-index*(*2*)[*of* $k - Suc\ i$ *remove-at-index* $k\ s'\ j\ x$] *insert-at-index*(*2*)[*of* $k - i$ *remove-at-index* $k\ s'\ j\ x$]
apply (*metis Suc.prems Suc-diff-le Suc-eq-plus1 Suc-leI* ‹$k - Suc\ i < length\ (insert\text{-}at\ (k - Suc\ i)\ x\ (remove\text{-}at\text{-}index\ k\ s')) - 1$› *diff-Suc-1 diff-Suc-Suc less-Suc-eq*)
by (*simp add*: *insert-at-index*(*4*) *nat-neq-iff*)
qed
show *all-axioms* α *sact spre uact upre*
by (*simp add*: *assms*(*6*))
qed
ultimately show *?case*
by *presburger*
qed
then have $\alpha\ (compute\ sact\ uact\ v0'\ (x\ \#\ ?s')) = \alpha\ (compute\ sact\ uact\ v0'\ s')$
using *i-def* **by** *force*
moreover have $\alpha\ (compute\ sact\ uact\ v0\ s1) = \alpha\ (compute\ sact\ uact\ v0'\ ?s')$

```
  proof (rule Suc(1))
    show n = length s1
      using Suc.hyps(2) ‹s = x # s1› by auto
    show α v0 = α v0'
      using assms(3) by auto
    show PRE-sequence spre upre s1 s1
      using PRE-sequenceE-rec(1) Suc.prems(4) ‹s = x # s1› by blast
    show possible-sequence (sargs − {# sarg #}) uargs s1
    proof (rule possible-sequenceI)
      show ⋀i. uargs i = map get-u (filter (is-Unique-i i) s1)
        by (metis (mono-tags, lifting) Shared Suc.hyps(2) Suc.prems(1) ‹s = x #
s1› action.disc(1) filter-remove is-Unique-def is-Unique-i-def le-numeral-extra(3)
nth-Cons-0 possible-sequence-def remove-at-index.simps(2) zero-less-Suc)
       show sargs − {#sarg#} = image-mset get-s (filter-mset is-Shared (mset
s1))
          using Shared Suc.prems(1) ‹s = x # s1› action.disc(1)[of sarg] ac-
tion.sel(1)[of sarg] add-mset-diff-bothsides diff-empty
       filter-mset-add-mset msed-map-invL mset.simps(2) possible-sequence-def[of
sargs uargs s]
        by simp
    qed

    show possible-sequence (sargs − {#sarg#}) uargs (remove-at-index k s')
    proof (rule possible-sequenceI)
     show ⋀i. uargs i = map get-u (filter (is-Unique-i i) (remove-at-index k s'))
      proof (rule list-ext)
        have filter is-Unique (remove-at-index k s') = filter is-Unique s'
          by (simp add: Shared ‹0 ≤ k ∧ k < length s' ∧ s' ! k = x› filter-remove
is-Unique-def)
        then show ⋀i. length (uargs i) = length (map get-u (filter (is-Unique-i i)
(remove-at-index k s')))
            by (metis Shared Suc.prems(2) ‹0 ≤ k ∧ k < length s' ∧ s' ! k = x›
action.disc(1) filter-remove is-Unique-def is-Unique-i-def possible-sequence-def)
        show ⋀i ia. 0 ≤ ia ∧ ia < length (uargs i) ⟹ uargs i ! ia = map get-u
(filter (is-Unique-i i) (remove-at-index k s')) ! ia
            by (metis Shared Suc.prems(2) ‹0 ≤ k ∧ k < length s' ∧ s' ! k = x›
action.disc(1) filter-remove is-Unique-def is-Unique-i-def possible-sequence-def)
      qed
      have sargs = image-mset get-s (filter-mset is-Shared (mset s'))
        using Suc.prems(2) possible-sequence-def by blast
       show sargs − {#sarg#} = image-mset get-s (filter-mset is-Shared (mset
(remove-at-index k s')))
      proof −
        have mset s' = mset (remove-at-index k s') + {# x #}
          using ‹0 ≤ k ∧ k < length s' ∧ s' ! k = x› mset-remove-index by blast
        then show ?thesis
          by (simp add: Shared ‹sargs = image-mset get-s (filter-mset is-Shared
(mset s'))›)
      qed
```

qed
show *PRE-sequence spre upre* (*remove-at-index k s'*) (*remove-at-index k s'*)
using *Suc.prems*(*5*) *PRE-sequence-removes* **by** *blast*
show *all-axioms* α *sact spre uact upre* **by** (*simp add*: *assms*(*6*))
qed
ultimately show *?thesis*
using ‹*all-axioms* α *sact spre uact upre*› *PRE-sequenceE-rec*(*2*) *Shared Suc.prems*(*4*) ‹*s* = *x* # *s1*› *abstract-pre.simps*(*1*) *compute.simps*(*2*) *sabstract*
by *fastforce*
next
case (*Unique ind uarg*)
let *?uargs* = *uargs ind*
have *hd ?uargs* = *uarg*
by (*metis Unique* ‹*s* = *x* # *s1*› *calculation*(*3*) *possible-sequence-where-is-unique*)
moreover have *?uargs* $\neq$ []
by (*metis* (*no-types*, *opaque-lifting*) *Suc.prems*(*1*) *Unique* ‹*s* = *x* # *s1*› *action.disc*(*2*) *action.sel*(*2*) *filter.simps*(*2*) *is-Unique-def is-Unique-i-def list.distinct*(*1*) *list.map-disc-iff possible-sequence-def*)
ultimately have *?uargs* = *uarg* # *tl ?uargs*
by *force*
then obtain *k* **where** $k \geq 0 \wedge k <$ *length* $s' \wedge s'$! $k = x$ $\bigwedge j.\ j \geq 0 \wedge j < k \Longrightarrow \neg$ *is-Unique-i ind* (s' ! j)
by (*metis Suc.prems*(*2*) *Unique* ‹*hd* (*uargs ind*) = *uarg*› ‹*uargs ind* $\neq$ []› *exists-index-in-sequence-unique*)
let *?s'* = *remove-at-index k s'*
have *length ?s'* = *length s'* − *1*
by (*simp add*: ‹$0 \leq k \wedge k <$ *length* $s' \wedge s'$! $k = x$› *remove-at-index*(*1*))
moreover have $k <$ *length s'*
by (*simp add*: ‹$0 \leq k \wedge k <$ *length* $s' \wedge s'$! $k = x$›)
then have *s'* = *insert-at k x ?s'*
by (*simp add*: ‹$0 \leq k \wedge k <$ *length* $s' \wedge s'$! $k = x$› *insert-remove-same-list*)
define *i* :: *nat* **where** *i* = *k*
have $i \geq 0 \wedge i \leq k \Longrightarrow \alpha$ (*compute sact uact v0'* (*insert-at* (*k* − *i*) *x ?s'*)) = α (*compute sact uact v0' s'*)
proof (*induct i*)
case *0*
then show *?case*
using ‹*s'* = *insert-at k x* (*remove-at-index k s'*)› **by** *auto*
next
case (*Suc i*)
then have α (*compute sact uact v0'* (*insert-at* (*k* − *i*) *x* (*remove-at-index k s'*))) = α (*compute sact uact v0' s'*)
using *Suc-leD* **by** *blast*
moreover have α (*compute sact uact v0'* (*insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s'*))) = α (*compute sact uact v0'* (*insert-at* (*k* − *i*) *x* (*remove-at-index k s'*)))
proof (*rule swap-works*)
show *length* (*insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s'*)) = *length* (*insert-at* (*k* − *i*) *x* (*remove-at-index k s'*))
by (*metis* (*no-types*, *lifting*) *Suc-pred'* ‹$0 \leq k \wedge k <$ *length* $s' \wedge s'$! $k =$

x› ‹*length* (*remove-at-index k s′*) = *length s′* − *1*› *insert-at-index*(*1*) *less-Suc-eq-le less-imp-diff-less order-le-less-trans*)

show *PRE-sequence spre upre* (*insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*)) (*insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*))

proof −

have *PRE-sequence spre upre* (*remove-at-index k s′*) (*remove-at-index k s′*)

using ‹*PRE-sequence spre upre s′ s′*›

by (*simp add*: *PRE-sequence-removes*)

then show *?thesis*

using ‹*abstract-pre spre upre x x*› *PRE-sequence-insert* **by** *blast*

qed

show α *v0′* = α *v0′* **by** *simp*

let *?k* = *k* − *Suc i*

show *?k* < *length* (*insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*)) − *1*

using *One-nat-def Suc.prems Suc-diff-Suc Suc-le-lessD* ‹*k* < *length s′*›

‹*length* (*remove-at-index k s′*) = *length s′* − *1*› ‹*s′* = *insert-at k x* (*remove-at-index k s′*)›

diff-le-self diff-zero insert-at-index(*1*)[*of k remove-at-index k s′ x*] *insert-at-index*(*1*)[*of k* − *Suc i remove-at-index k s′ x*]

less-Suc-eq-le order-le-less-trans

by *simp*

show *insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*) ! *?k* = *insert-at* (*k* − *i*) *x* (*remove-at-index k s′*) ! (*?k* + *1*)

proof −

have *insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*) ! (*k* − *Suc i*) = *x*

by (*metis Suc-pred′* ‹*k* < *length s′*› ‹*length* (*remove-at-index k s′*) = *length s′* − *1*› *bot-nat-0.extremum diff-self-eq-0 insert-at-index*(*3*) *less-Suc-eq-le less-imp-diff-less*)

moreover have *insert-at* (*k* − *i*) *x* (*remove-at-index k s′*) ! (*k* − *i*) = *x*

by (*metis Suc-pred′* ‹*k* < *length s′*› ‹*length* (*remove-at-index k s′*) = *length s′* − *1*› *bot-nat-0.extremum insert-at-index*(*3*) *less-Suc-eq-le less-imp-diff-less less-nat-zero-code not-gr-zero*)

ultimately show *?thesis*

by (*simp add*: *Suc.prems Suc-diff-Suc Suc-le-lessD*)

qed

have *insert-at* (*k* − *i*) *x* (*remove-at-index k s′*) ! (*k* − *Suc i*) = *remove-at-index k s′* ! (*k* − *Suc i*)

by (*metis* (*no-types*, *lifting*) *Suc.prems Suc-diff-Suc Suc-eq-plus1 Suc-leI Suc-le-lessD* ‹*k* < *length s′*› ‹*length* (*remove-at-index k s′*) = *length s′* − *1*› *add-leE insert-at-index*(*2*) *le-add-diff-inverse2 le-add-same-cancel2 lessI less-Suc-eq-le*)

then

show *insert-at* (*k* − *i*) *x* (*remove-at-index k s′*) ! *?k* = *insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*) ! (*?k* + *1*)

using *One-nat-def Suc.prems Suc-diff-Suc* ‹*k* − *Suc i* < *length* (*insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*)) − *1*›

‹*k* < *length s′*› ‹*length* (*remove-at-index k s′*) = *length s′* − *1*›

add-diff-cancel-right′
add-leE diff-zero insert-at-index(*1*)[*of k* − *Suc i remove-at-index k s′ x*]
insert-at-index(*4*)[*of k* − *Suc i remove-at-index k s′*]
le-add-diff-inverse2 less-Suc-eq-le
less-add-same-cancel1 less-diff-conv order-le-less-trans zero-less-one
by *simp*

have *insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*) ! (*k* − *Suc i* + *1*) = *remove-at-index k s′* ! (*k* − *Suc i* + *1* − *1*)
using ‹*insert-at* (*k* − *i*) *x* (*remove-at-index k s′*) ! (*k* − *Suc i*) = *insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*) ! (*k* − *Suc i* + *1*)› ‹*insert-at* (*k* − *i*) *x* (*remove-at-index k s′*) ! (*k* − *Suc i*) = *remove-at-index k s′* ! (*k* − *Suc i*)› **by** *auto*
then have ¬ *is-Unique-i ind* (*insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*) ! (*?k* + *1*))
by (*metis One-nat-def Suc.prems Suc-le-lessD* ‹⋀*j*. *0* ≤ *j* ∧ *j* < *k* ⟹ ¬ *is-Unique-i ind* (*s′* ! *j*)› ‹*k* < *length s′*› *add-diff-cancel-right′ add-leE diff-Suc-less le-add2 le-add-same-cancel2 plus-1-eq-Suc remove-at-index*(*2*))
then show ¬ (∃ *k′*. *is-Unique-i k′* (*insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*) ! (*k* − *Suc i*)) ∧
is-Unique-i k′ (*insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*) ! (*k* − *Suc i* + *1*)))
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc.prems Suc-diff-Suc Unique* ‹*k* < *length s′*› ‹*length* (*remove-at-index k s′*) = *length s′* − *1*› *action.sel*(*2*) *diff-zero insert-at-index*(*3*) *is-Unique-i-def le-add2 le-add-diff-inverse le-add-same-cancel2 less-Suc-eq-le order-le-less-trans*)
show ⋀*j*. *0* ≤ *j* ∧ *j* < *length* (*insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*)) ∧ *j* ≠ *?k* ∧ *j* ≠ *?k* + *1* ⟹
insert-at (*k* − *Suc i*) *x* (*remove-at-index k s′*) ! *j* = *insert-at* (*k* − *i*) *x* (*remove-at-index k s′*) ! *j*
proof −
fix *j* **assume** *0* ≤ *j* ∧ *j* < *length* (*insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*)) ∧ *j* ≠ *?k* ∧ *j* ≠ *?k* + *1*
moreover have *k* − *Suc i* ≤ *length* (*remove-at-index k s′*)
using ‹*0* ≤ *k* ∧ *k* < *length s′* ∧ *s′* ! *k* = *x*› ‹*length* (*remove-at-index k s′*) = *length s′* − *1*› **by** *force*
moreover have *k* − *i* ≤ *length* (*remove-at-index k s′*)
using ‹*k* < *length s′*› ‹*length* (*remove-at-index k s′*) = *length s′* − *1*› **by** *linarith*
then show *insert-at* (*k* − *Suc i*) *x* (*remove-at-index k s′*) ! *j* = *insert-at* (*k* − *i*) *x* (*remove-at-index k s′*) ! *j*
apply (*cases j* < *k* − *i*)
apply (*metis Suc.prems Suc-diff-Suc Suc-le-lessD calculation*(*1*) *calculation*(*2*) *insert-at-index*(*2*) *less-Suc-eq*)
by (*metis Suc.prems Suc-diff-Suc Suc-eq-plus1 Suc-le-lessD calculation*(*1*) *calculation*(*2*) *insert-at-index*(*1*) *insert-at-index*(*4*) *linorder-le-less-linear linorder-neqE-nat*)
qed
show *all-axioms α sact spre uact upre* **by** (*simp add*: *assms*(*6*))
qed

ultimately show *?case*
by *presburger*
qed
then have α (*compute sact uact v0′* (*x* # *?s′*)) = α (*compute sact uact v0′ s′*)
using *i-def* **by** *force*
moreover have α (*compute sact uact v0 s1*) = α (*compute sact uact v0′ ?s′*)
proof (*rule Suc*(*1*))
show *all-axioms* α *sact spre uact upre* **by** (*simp add*: *assms*(*6*))
show *n* = *length s1*
using *Suc.hyps*(*2*) ‹*s* = *x* # *s1*› **by** *auto*
show α *v0* = α *v0′*
using *assms*(*3*) **by** *auto*
show *PRE-sequence spre upre s1 s1*
using *Suc.prems*(*4*) ‹*s* = *x* # *s1*› *PRE-sequenceE-rec*(*1*) **by** *blast*
show *possible-sequence sargs* (*tail-kth uargs ind*) *s1*
proof (*rule possible-sequenceI*)
show $\bigwedge$*i*. *tail-kth uargs ind i* = *map get-u* (*filter* (*is-Unique-i i*) *s1*)
proof −
fix *i* **show** *tail-kth uargs ind i* = *map get-u* (*filter* (*is-Unique-i i*) *s1*)
proof (*cases i* = *ind*)
case *True*
then have *tail-kth uargs ind i* = *tl ?uargs*
by (*simp add*: *tail-kth-def*)
then show *?thesis* **using** *exists-index-in-sequence-unique*[*of uarg uargs ind sargs s*]
by (*metis Suc.prems*(*1*) *Unique* ‹*hd* (*uargs ind*) = *uarg*› ‹*s* = *x* # *s1*› ‹*uargs ind* ≠ []› *action.disc*(*2*) *action.sel*(*2*) *is-Unique-def is-Unique-i-def le-eq-less-or-eq nth-Cons-0 possible-sequence-def remove-at-index.simps*(*2*))
next
case *False*
then show *?thesis*
using *Suc.hyps*(*2*) *Suc.prems*(*1*) *Unique* ‹*s* = *x* # *s1*› *action.sel*(*2*) *filter-remove*
fun-upd-apply is-Unique-i-def le-numeral-extra(*3*) *nth-Cons-0*[*of x s1*] *possible-sequence-def*[*of sargs uargs s*]
remove-at-index.simps(*2*)[*of x s1*] *tail-kth-def zero-less-Suc*
by *metis*
qed
qed
show *sargs* = *image-mset get-s* (*filter-mset is-Shared* (*mset s1*))
by (*metis Suc.prems*(*1*) *Unique* ‹*s* = *x* # *s1*› *action.disc*(*2*) *filter-mset-add-mset mset.simps*(*2*) *possible-sequence-def*)
qed
show *possible-sequence sargs* (*tail-kth uargs ind*) (*remove-at-index k s′*)
proof (*rule possible-sequenceI*)
show $\bigwedge$*i*. *tail-kth uargs ind i* = *map get-u* (*filter* (*is-Unique-i i*) (*remove-at-index k s′*))
using *Suc.prems*(*1*) *Suc.prems*(*2*) *Unique* ‹*0* ≤ *k* ∧ *k* < *length s′* ∧ *s′* ! *k* = *x*› ‹$\bigwedge$*j*. *0* ≤ *j* ∧ *j* < *k* ⟹ ¬ *is-Unique-i ind* (*s′* ! *j*)›

‹possible-sequence sargs (tail-kth uargs ind) s1› ‹s = x # s1› action.sel(2) filter.simps(2)
filter-remove fun-upd-same is-Unique-i-def possible-sequence-def[of sargs tail-kth uargs ind s1]
possible-sequence-def[of sargs uargs s] possible-sequence-def[of sargs uargs s']
remove-at-index-filter tail-kth-def
by *metis*
show *sargs = image-mset get-s (filter-mset is-Shared (mset (remove-at-index k s')))*
by *(metis Suc.prems(2) Unique ‹0 ≤ k ∧ k < length s' ∧ s' ! k = x› action.disc(2) filter-remove mset-filter possible-sequence-def)*
qed
show *PRE-sequence spre upre (remove-at-index k s') (remove-at-index k s')*
using *PRE-sequence-removes Suc.prems(5)* **by** *auto*
qed
ultimately show *?thesis*
using *Unique ‹abstract-pre spre upre x x› ‹s = x # s1› abstract-pre.simps(2)[of]*
assms(6) compute.simps(3)[of sact uact] uabstract[of α sact spre uact upre]
by *metis*
qed
qed

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')*
using *assms*
proof *(induct s' arbitrary: s v0 v0')*
case *Nil*
then show *?case*
by *(simp add: PRE-sequence-def)*
next
case *(Cons act' s')*
then show *?case*
proof *(cases act')*
case *(Shared sarg')*
then obtain *sarg s0* **where** *s = Shared sarg # s0 spre sarg sarg' PRE-sequence spre upre s0 s'*
by *(metis Cons.prems(1) PRE-sequenceE-rec(1) PRE-sequenceE-rec(2) PRE-sequence-def abstract-pre.simps(1) abstract-pre.simps(3) action.exhaust length-0-conv neq-Nil-conv)*
then show *?thesis*
using *Cons.hyps Cons.prems(2) Cons.prems(3) Shared sabstract* **by** *fastforce*
next
case *(Unique k uarg')*
then obtain *uarg s0* **where** *s = Unique k uarg # s0 upre k uarg uarg' PRE-sequence spre upre s0 s'*

by (*metis Cons.prems(1) PRE-sequenceE-rec(1) PRE-sequenceE-rec(2) PRE-sequence-def abstract-pre.simps(2) abstract-pre.simps(4) action.exhaust length-0-conv neq-Nil-conv*)
then show *?thesis*
using *Cons.hyps Cons.prems(2) Unique assms(3) uabstract* **by** *fastforce*
qed
qed

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 (*rule PRE-sequenceI*)
show *length s′ = length s′* **by** *simp*
fix *i* **assume** $0 \leq i \wedge i <$ *length s′*
then show *abstract-pre spre upre (s′ ! i) (s′ ! i)*
proof (*cases s′ ! i*)
case (*Shared sarg′*)
then have *Shared sarg′* ∈# *filter-mset is-Shared (mset s′)*
using ‹$0 \leq i \wedge i <$ *length s′*› *nth-mem-mset* **by** *fastforce*
then have *sarg′* ∈# *sargs′*
by (*metis (mono-tags, lifting) action.sel(1) assms(2) imageI possible-sequence-def set-image-mset*)
moreover obtain *ms* **where** *image-mset fst ms = sargs* ∧ *image-mset snd ms = sargs′* ∧ (∀ *x* ∈# *ms. spre (fst x) (snd x)*)
using *PRE-shared-def assms(3)* **by** *blast*
then obtain *x* **where** *x* ∈# *ms snd x = sarg′*
using *calculation* **by** *fastforce*
then show *?thesis*
using *Shared* ‹*image-mset fst ms = sargs* ∧ *image-mset snd ms = sargs′* ∧ (∀ *x*∈#*ms. spre (fst x) (snd x)*)› *spre-refl*
by (*metis abstract-pre.simps(1) assms(5)*)
next
case (*Unique k uarg′*)
then have *Unique k uarg′* ∈ *set (filter is-Unique s′)*
by (*metis* ‹$0 \leq i \wedge i <$ *length s′*› *is-Unique-def action.disc(2) filter-set member-filter nth-mem*)
then have *uarg′* ∈ *set (map get-u (filter (is-Unique-i k) s′))*
by (*metis (no-types, lifting) action.sel(2) action.sel(3) filter-set image-eqI is-Unique-i-def list.set-map member-filter*)
then obtain *i* **where** $i \geq 0 \wedge i <$ *length (uargs′ k)* ∧ *uarg′ = (uargs′ k) ! i*
by (*metis assms(2) gr-implies-not-zero in-set-conv-nth linorder-le-less-linear possible-sequence-def*)
then have *upre k ((uargs k) ! i) ((uargs′ k) ! i)*
using *PRE-unique-def assms(4)* **by** *blast*
then show *?thesis*
using *Unique* ‹$0 \leq i \wedge i <$ *length (uargs′ k)* ∧ *uarg′ = uargs′ k ! i*› *assms(5)*

upre-refl **by** *fastforce*
  qed
qed

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 −
  obtain *s''* **where** *possible-sequence sargs' uargs' s''* ∧ *PRE-sequence spre upre s s''*
    using *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *exists-aligned-sequence* **by** *blast*
  have α (*compute sact uact v0' s''*) = α (*compute sact uact v0' s'*)
  proof (*rule it-all-commutes*)
    show *possible-sequence sargs' uargs' s''*
      by (*simp add*: ‹*possible-sequence sargs' uargs' s''* ∧ *PRE-sequence spre upre s s''*›)
    show *possible-sequence sargs' uargs' s'*
      by (*simp add*: *assms*(*2*))
    show α *v0'* = α *v0'*
      by *simp*
    show *PRE-sequence spre upre s'' s''*
      using ‹*possible-sequence sargs' uargs' s''* ∧ *PRE-sequence spre upre s s''*› *PRE-sequence-refl assms*(*6*) **by** *blast*
    show *PRE-sequence spre upre s' s'*
     using *simple-possible-PRE-seq assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*6*) **by** *blast*
    show *all-axioms* α *sact spre uact upre*
      using *assms*(*6*) **by** *auto*
  qed
  moreover have α (*compute sact uact v0' s''*) = α (*compute sact uact v0 s*)
   using *PRE-sequence-same-abstract* ‹*possible-sequence sargs' uargs' s''* ∧ *PRE-sequence spre upre s s''*› *assms*(*1*) *assms*(*5*) *assms*(*6*) **by** *metis*
  ultimately show *?thesis*
    by *auto*
qed

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

inductive *reachable-value* :: ('*v* ⇒ '*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*
  using *assms*
proof (*induct rule*: *reachable-value.induct*)
  case (*Self sact uact v0*)
  have *possible-sequence* {#} (λ*k*. []) [] ∧ *v0* = *compute sact uact v0* []
    by (*simp add*: *possible-sequenceI*)
  then show *?case* **by** *blast*
next
  case (*SharedStep sact uact v0 sargs uargs v1 sarg*)
  then obtain *s* **where** *possible-sequence sargs uargs s* ∧ *v1* = *compute sact uact v0 s* **by** *blast*
  let *?s* = *Shared sarg* # *s*
  have *possible-sequence* (*sargs* + {#*sarg*#}) *uargs ?s*
  proof (*rule possible-sequenceI*)
    show ⋀*i*. *uargs i* = *map get-u* (*filter* (*is-Unique-i i*) (*Shared sarg* # *s*))
      by (*metis* ‹*possible-sequence sargs uargs s* ∧ *v1* = *compute sact uact v0 s*› *action.disc*(*1*) *filter.simps*(*2*) *is-Unique-def is-Unique-i-def possible-sequence-def*)
    show *sargs* + {#*sarg*#} = *image-mset get-s* (*filter-mset is-Shared* (*mset* (*Shared sarg* # *s*)))
      using ‹*possible-sequence sargs uargs s* ∧ *v1* = *compute sact uact v0 s*› *possible-sequence-def* **by** *auto*
  qed
  then show *?case*
    using ‹*possible-sequence sargs uargs s* ∧ *v1* = *compute sact uact v0 s*› **by** *auto*
next
  case (*UniqueStep sact uact v0 sargs uargs v1 k uarg*)
  then obtain *s* **where** *possible-sequence sargs uargs s* ∧ *v1* = *compute sact uact v0 s* **by** *blast*
  let *?s* = *Unique k uarg* # *s*
  have *possible-sequence sargs* (*uargs*(*k* := *uarg* # *uargs k*)) *?s*
  proof (*rule possible-sequenceI*)
    show ⋀*i*. (*uargs*(*k* := *uarg* # *uargs k*)) *i* = *map get-u* (*filter* (*is-Unique-i i*) (*Unique k uarg* # *s*))
    proof −
      fix *i* **show** (*uargs*(*k* := *uarg* # *uargs k*)) *i* = *map get-u* (*filter* (*is-Unique-i i*) (*Unique k uarg* # *s*))
      proof (*cases i* = *k*)
        case *True*
        then show *?thesis*
          using *Cons-eq-map-conv* ‹*possible-sequence sargs uargs s* ∧ *v1* = *compute sact uact v0 s*›

action.disc(*2*) *action.sel*(*2*) *action.sel*(*3*) *filter.simps*(*2*) *fun-upd-same* *is-Unique-def*
is-Unique-i-def *possible-sequence-def*[*of sargs uargs s*]
by *fastforce*
next
case *False*
then show *?thesis*
by (*metis* ‹*possible-sequence sargs uargs s* ∧ *v1* = *compute sact uact v0 s*› *action.sel*(*2*) *filter.simps*(*2*) *fun-upd-other* *is-Unique-i-def* *possible-sequence-def*)
qed
qed
show *sargs* = *image-mset get-s* (*filter-mset is-Shared* (*mset* (*Unique k uarg* # *s*)))
using ‹*possible-sequence sargs uargs s* ∧ *v1* = *compute sact uact v0 s*› *possible-sequence-def* **by** *force*
qed
then show *?case* **using** ‹*possible-sequence sargs uargs s* ∧ *v1* = *compute sact uact v0 s*›
by (*metis compute.simps*(*3*))
qed

lemma *PRE-shared-simpler-implies*:
assumes *PRE-shared-simpler spre a b*
shows *PRE-shared spre a b*
using *assms*
proof (*induct rule*: *PRE-shared-simpler.induct*)
case (*Empty spre*)
then show *?case*
by (*simp add*: *PRE-shared-def*)
next
case (*Step spre a b xa xb*)
then obtain *ms* **where** *image-mset fst ms* = *a* ∧ *image-mset snd ms* = *b* ∧ (∀ *x*∈#*ms*. *spre* (*fst x*) (*snd x*))
by (*metis PRE-shared-def*)
then have *image-mset fst* (*ms* + {# (*xa*, *xb*) #}) = (*a* + {#*xa*#}) ∧ *image-mset snd* (*ms* + {# (*xa*, *xb*) #}) = (*b* + {#*xb*#}) ∧ (∀ *x*∈#(*ms* + {# (*xa*, *xb*) #}). *spre* (*fst x*) (*snd x*))
using *Step.hyps*(*3*) **by** *auto*
then show *?case* **using** *PRE-shared-def* **by** *blast*
qed

The following theorem corresponds to Lemma 4.2 in the paper.

theorem *main-result*:
assumes *reachable-value sact uact v0 sargs uargs v*
and *reachable-value sact uact v0′ sargs′ uargs′ v′*
and *PRE-shared-simpler spre sargs sargs′*
and ⋀*k*. *PRE-unique* (*upre k*) (*uargs k*) (*uargs′ k*)
and α *v0* = α *v0′*
and *all-axioms* α *sact spre uact upre*

shows $\alpha\ v = \alpha\ v'$
proof −
obtain *s s'* **where** *possible-sequence sargs uargs s* ∧ *v* = *compute sact uact v0 s* *possible-sequence sargs' uargs' s'* ∧ *v'* = *compute sact uact v0' s'*
using *assms*(*1*) *assms*(*2*) *reachable-then-possible-sequence-and-compute*
by *metis*
then show *?thesis*
by (*meson PRE-shared-simpler-implies assms*(*3*) *assms*(*4*) *assms*(*5*) *assms*(*6*) *main-lemma*)
qed

end

4.2 Consistency

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

theory *Guards*
imports *StateModel CommCSL AbstractCommutativity*
begin

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

definition *reachable* :: ('*i*, '*a*, '*v*) *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 ⋀*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*
by (*metis assms reachable-def*)

lemma *reachableE*:
assumes *reachable scont v0 h*
and *get-gs h* = *Some* (*pwrite*, *sargs*)
and ⋀*k*. *get-gu h k* = *Some* (*uargs k*)
shows *reachable-value* (*saction scont*) (*uaction scont*) *v0 sargs uargs* (*view scont* (*normalize* (*get-fh h*)))
by (*meson assms reachable-def*)

definition *all-guards* :: ('*i*, '*a*) *heap* ⇒ *bool* **where**

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

lemma *no-guardI*:
assumes *get-gs h* = *None*
and $\bigwedge$*k. get-gu h k* = *None*
shows *no-guard h*
using *assms*(*1*) *assms*(*2*) *no-guard-def* **by** *blast*

definition *semi-consistent* :: ('*i*, '*a*, '*v*) *single-context* $\Rightarrow$ '*v* $\Rightarrow$ ('*i*, '*a*) *heap* $\Rightarrow$ *bool*
where
semi-consistent Γ *v0 h* $\longleftrightarrow$ *all-guards h* $\wedge$ *reachable* Γ *v0 h*

lemma *semi-consistentE*:
assumes *semi-consistent* Γ *v0 h*
shows $\exists$ *sargs uargs. get-gs h* = *Some* (*pwrite*, *sargs*) $\wedge$ ($\forall$ *k. get-gu h k* = *Some* (*uargs k*))
$\wedge$ *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs uargs* (*view* Γ (*normalize* (*get-fh h*)))
proof −
let *?uargs* = λ*k.* (*SOME x. get-gu h k* = *Some x*)
have $\bigwedge$*k. get-gu h k* = *Some* (*?uargs k*)
proof −
fix *k* **have** $\exists$ *x. get-gu h k* = *Some x*
by (*meson all-guards-def assms option.exhaust-sel semi-consistent-def*)
then show *get-gu h k* = *Some* (*?uargs k*)
by *fastforce*
qed
moreover obtain *sargs* **where** *get-gs h* = *Some* (*pwrite*, *sargs*)
by (*meson all-guards-def assms semi-consistent-def*)
ultimately have *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs ?uargs* (*view* Γ (*normalize* (*get-fh h*)))
by (*meson assms reachableE semi-consistent-def*)
then show *?thesis*
using ‹$\bigwedge$*k. get-gu h k* = *Some* (*SOME x. get-gu h k* = *Some x*)› ‹*get-gs h* = *Some* (*pwrite*, *sargs*)› **by** *fastforce*
qed

lemma *semi-consistentI*:
assumes *all-guards h*
and *reachable* Γ *v0 h*
shows *semi-consistent* Γ *v0 h*
by (*simp add*: *assms*(*1*) *assms*(*2*) *semi-consistent-def*)

lemma *no-guard-then-smaller-same*:
assumes *Some h* = *Some a* $\oplus$ *Some b*
and *no-guard h*
shows *no-guard a*
proof (*rule no-guardI*)
show *get-gs a* = *None*

by (*metis add-gs.elims assms*(*1*) *assms*(*2*) *no-guard-def option.simps*(*3*) *plus-extract*(*2*))
fix *k*
have *get-gu h k* = *None*
by (*meson assms*(*2*) *no-guard-def*)
then show *get-gu a k* = *None*
by (*metis assms*(*1*) *full-uguard-sum-same option.exhaust*)
qed

lemma *all-guardsI*:
assumes $\bigwedge k$. *get-gu h k* $\neq$ *None*
and $\exists$ *v*. *get-gs h* = *Some* (*pwrite*, *v*)
shows *all-guards h*
using *all-guards-def assms*(*1*) *assms*(*2*) **by** *blast*

lemma *all-guards-same*:
assumes *all-guards a*
and *Some h* = *Some a* $\oplus$ *Some b*
shows *all-guards h*
proof (*rule all-guardsI*)
show $\exists$ *v*. *get-gs h* = *Some* (*pwrite*, *v*)
using *all-guards-def assms*(*1*) *assms*(*2*) *full-sguard-sum-same* **by** *blast*
fix *k* **have** *get-gu a k* $\neq$ *None*
by (*meson all-guards-def assms*(*1*))
then show *get-gu h k* $\neq$ *None*
apply (*cases get-gu b k*)
apply (*metis assms*(*2*) *full-uguard-sum-same not-Some-eq*)
by (*metis assms*(*2*) *full-uguard-sum-same option.discI plus-comm*)
qed

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

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

lemma *remove-guards-smaller*:
h $\succeq$ *remove-guards h*
proof −
have *remove-guards h* ## (*Map.empty*, *get-gs h*, *get-gu h*)
proof (*rule compatibleI*)
show *compatible-fract-heaps* (*get-fh* (*remove-guards h*)) (*get-fh* (*Map.empty*, *get-gs h*, *get-gu h*))
using *compatible-fract-heapsI* **by** *force*
show $\bigwedge k$. *get-gu* (*remove-guards h*) *k* = *None* $\vee$ *get-gu* (*Map.empty*, *get-gs h*, *get-gu h*) *k* = *None*
by (*simp add*: *empty-unique-def remove-guards-def*)
show $\bigwedge p\ p'$. *get-gs* (*remove-guards h*) = *Some p* $\wedge$ *get-gs* (*Map.empty*, *get-gs h*, *get-gu h*) = *Some p'* $\Longrightarrow$ *pgte pwrite* (*padd* (*fst p*) (*fst p'*))
by (*simp add*: *remove-guards-def*)

qed
then obtain x **where** *Some* x = *Some* (*remove-guards* h) ⊕ *Some* (*Map.empty*, *get-gs* h, *get-gu* h)
by *auto*
moreover have $x = h$
proof (*rule heap-ext*)
show *get-fh* x = *get-fh* h
by (*metis add-fh-map-empty add-get-fh calculation fst-conv get-fh.elims remove-guards-def*)
show *get-gs* x = *get-gs* h
by (*metis calculation fst-eqD get-gs.elims plus-comm remove-guards-def snd-eqD sum-gs-one-none*)
show *get-gu* x = *get-gu* h
proof (*rule ext*)
fix k
have *get-gu* (*remove-guards* h) k = *None*
by (*simp add*: *empty-unique-def remove-guards-def*)
then show *get-gu* x k = *get-gu* h k
by (*metis* (*mono-tags*, *lifting*) *add-gu-def add-gu-single.simps*(*1*) *calculation get-gu.elims plus-extract*(*3*) *snd-eqD*)
qed
qed
ultimately show *?thesis*
using *larger-def* **by** *blast*
qed

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
using *assms*(*1*) *assms*(*2*) *no-guard-def sum-gs-one-none* **apply** *blast*
proof (*rule ext*)
fix k
have *get-gu* c k = *None*
by (*meson assms*(*2*) *no-guard-def*)
then show *get-gu* a k = *get-gu* b k
by (*metis* (*no-types*, *lifting*) *add-gu-def add-gu-single.simps*(*1*) *assms*(*1*) *plus-comm plus-extract*(*3*))
qed

lemma *full-guard-comp-then-no*:
assumes a ## b
and *all-guards* a
shows *no-guard* b
proof (*rule no-guardI*)
show $\bigwedge k$. *get-gu* b k = *None*
by (*meson all-guards-def assms*(*1*) *assms*(*2*) *compatible-def*)
show *get-gs* b = *None*

proof (*rule ccontr*)
  assume *get-gs b* $\neq$ *None*
  then obtain *gb* **where** *get-gs b* = *Some gb*
    by *blast*
  moreover obtain *v* **where** *get-gs a* = *Some* (*pwrite*, *v*)
    by (*meson all-guards-def assms*(*2*))
  moreover have *pgt* (*padd pwrite* (*fst gb*)) *pwrite*
    using *sum-larger* **by** *auto*
  ultimately show *False*
    by (*metis assms*(*1*) *compatible-def fst-eqD not-pgte-charact*)
qed
qed

lemma *sum-of-no-guards*:
  assumes *no-guard a*
    and *no-guard b*
    and *Some x* = *Some a* $\oplus$ *Some b*
  shows *no-guard x*
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *no-guard-def no-guard-remove*(*1*) *no-guard-remove*(*2*))

lemma *no-guard-remove-guards*:
  no-guard (*remove-guards h*)
  by (*simp add*: *empty-unique-def no-guard-def remove-guards-def*)

lemma *get-fh-remove-guards*:
  get-fh (*remove-guards h*) = *get-fh h*
  by (*simp add*: *remove-guards-def*)

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

lemma *pair-satI*:
  assumes $\bigwedge s\ h\ s'\ h'.\ (s, h) \in S \wedge (s', h') \in S' \Longrightarrow (s, h), (s', h') \models Q$
  shows *pair-sat* S S' Q
  by (*simp add*: *assms pair-sat-def*)

lemma *pair-sat-smallerI*:
  assumes $\bigwedge \sigma\ \sigma'.\ \sigma \in S \wedge \sigma' \in S' \Longrightarrow \sigma, \sigma' \models Q$
  shows *pair-sat* S S' Q
  by (*simp add*: *assms pair-sat-def*)

lemma *pair-satE*:
  assumes *pair-sat* S S' Q
    and $(s, h) \in S \wedge (s', h') \in S'$
  shows $(s, h), (s', h') \models Q$
  using *assms*(*1*) *assms*(*2*) *pair-sat-def* **by** *blast*

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

$(store \times ('i, 'a)\ heap)\ set$ **where**
add-states S1 S2 = $\{(s, H)$ |*s H h1 h2*. *Some H* = *Some h1* $\oplus$ *Some h2* $\wedge$ $(s, h1) \in S1 \wedge (s, h2) \in S2\}$

lemma *add-states-sat-star*:
assumes *pair-sat SA SA′ A*
and *pair-sat SB SB′ B*
shows *pair-sat* (*add-states SA SB*) (*add-states SA′ SB′*) (*Star A B*)
proof (*rule pair-satI*)
fix *s h s′ h′*
assume *asm0*: $(s, h) \in$ *add-states SA SB* $\wedge$ $(s', h') \in$ *add-states SA′ SB′*
then obtain *ha hb ha′ hb′* **where** $(s, ha) \in SA$ $(s, hb) \in SB$ $(s', ha') \in SA'$ $(s', hb') \in SB'$
Some h = *Some ha* $\oplus$ *Some hb* *Some h′* = *Some ha′* $\oplus$ *Some hb′*
using *add-states-def*[*of SA SB*] *add-states-def*[*of SA′ SB′*] *fst-eqD mem-Collect-eq snd-conv*
by *auto*
then show $(s, h), (s', h') \models$ *Star A B*
by (*meson assms*(*1*) *assms*(*2*) *hyper-sat.simps*(*4*) *pair-sat-def*)
qed

lemma *add-states-subset*:
assumes $S1 \subseteq S1'$
shows *add-states S1 S2* $\subseteq$ *add-states S1′ S2*
proof
fix *x* **assume** $x \in$ *add-states S1 S2*
then show $x \in$ *add-states S1′ S2*
using *add-states-def*[*of S1 S2*] *add-states-def*[*of S1′ S2*] *assms mem-Collect-eq*[*of x*] *subsetD*[*of S1 S1′*]
by *blast*
qed

lemma *add-states-comm*:
add-states S1 S2 = *add-states S2 S1*
proof −
have $\bigwedge$*S1 S2*. *add-states S1 S2* $\subseteq$ *add-states S2 S1*
proof −
fix *S1 S2*
show *add-states S1 S2* $\subseteq$ *add-states S2 S1*
proof
fix *x* **assume** $x \in$ *add-states S1 S2*
then obtain *h1 h2* **where** *Some* (*snd x*) = *Some h1* $\oplus$ *Some h2* (*fst x*, *h1*) $\in S1$ (*fst x*, *h2*) $\in S2$
using *add-states-def*[*of S1 S2*] *fst-conv mem-Collect-eq*[*of x*] *snd-eqD*
by *auto*
moreover have *Some* (*snd x*) = *Some h2* $\oplus$ *Some h1*
by (*simp add*: *calculation*(*1*) *plus-comm*)
ultimately show $x \in$ *add-states S2 S1*
using *add-states-def*[*of S2 S1*] *mem-Collect-eq*[*of x*] *surjective-pairing*[*of x*]

by *blast*
  qed
 qed
 then show *?thesis* **by** *blast*
qed

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

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 −
 obtain *a1′ a2′ j1′ j2′* **where** *Some x1 = Some a1′ ⊕ Some j1′*
 Some x2 = Some a2′ ⊕ Some j2′ (s1, j1′), (s2, j2′) ⊨ J (s1, a1′), (s2, a2′) ⊨ A
  using *assms(3) hyper-sat.simps(4)* **by** *blast*
 have *j1 = j1′ ∧ j2 = j2′*
  using *assms(5)*
 proof *(rule preciseE)*
  show *x1 ⪰ j1′ ∧ x1 ⪰ j1 ∧ x2 ⪰ j2′ ∧ x2 ⪰ j2*
   by *(metis ‹Some x1 = Some a1′ ⊕ Some j1′› ‹Some x2 = Some a2′ ⊕ Some j2′› assms(1) assms(2) larger-def plus-comm)*
  show *(s1, j1′), (s2, j2′) ⊨ J ∧ (s1, j1), (s2, j2) ⊨ J*
   by *(simp add: ‹(s1, j1′), (s2, j2′) ⊨ J› assms(4))*
 qed
 then have *a1 = a1′ ∧ a2 = a2′*
  using *‹Some x1 = Some a1′ ⊕ Some j1′› ‹Some x2 = Some a2′ ⊕ Some j2′› addition-cancellative assms(1) assms(2)* **by** *blast*
 then show *?thesis*
  using *‹(s1, a1′), (s2, a2′) ⊨ A›* **by** *blast*
qed

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 *(rule heap-ext)*
 show *get-gu h = get-gu h′*
  apply *(rule ext)*
  by *(metis assms(1) assms(2) no-guard-def)*
 show *get-gs h = get-gs h′*
  by *(metis assms(1) assms(2) no-guard-def)*
 show *get-fh h = get-fh h′*

proof (*rule ext*)
fix *l* **show** *get-fh h l* = *get-fh h′ l*
apply (*cases get-fh h l*)
apply (*metis FractionalHeap.normalize-eq*(*1*) *assms*(*3*))
apply (*cases get-fh h′ l*)
apply (*metis FractionalHeap.normalize-eq*(*1*) *assms*(*3*))
by (*metis FractionalHeap.normalize-def apply-opt.simps*(*2*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *full-ownership-def prod.collapse*)
qed
qed

lemma *get-fh-same-then-remove-guards-same*:
assumes *get-fh a* = *get-fh b*
shows *remove-guards a* = *remove-guards b*
by (*metis assms remove-guards-def*)

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 −
have *remove-guards a ## remove-guards b*
by (*metis* (*no-types*, *lifting*) *assms compatible-def compatible-eq get-fh-remove-guards no-guard-def no-guard-remove-guards option.distinct*(*1*))
then obtain *y* **where** *Some y* = *Some* (*remove-guards a*) ⊕ *Some* (*remove-guards b*)
by *auto*
moreover have *remove-guards x* = *y*
by (*metis* (*no-types*, *lifting*) ‹*remove-guards a ## remove-guards b*› *add-get-fh assms calculation get-fh-remove-guards get-gu.simps no-guard-def no-guard-remove*(*1*) *no-guard-remove*(*2*) *no-guard-remove-guards option.inject plus.simps*(*3*) *plus-extract*(*2*) *remove-guards-def snd-eqD*)
ultimately show *?thesis* **by** *blast*
qed

lemma *no-guard-smaller*:
assumes *a* ⪰ *b*
shows *remove-guards a* ⪰ *remove-guards b*
using *assms larger-def remove-guards-sum* **by** *blast*

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*
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *compatible-def compatible-fract-heapsI*

no-guard-def option.simps(3))

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 −
obtain *x* **where** *Some x* = *Some h* ⊕ *Some* (*Map.empty*, *Some* (*pwrite*, {#}), (λ-. *Some* []))
by (*simp add*: *assms no-guard-map-empty-compatible*)
moreover have *add-empty-guards h* = *x*
proof (*rule heap-ext*)
show *get-fh* (*add-empty-guards h*) = *get-fh x*
by (*metis add-empty-guards-def add-fh-map-empty add-get-fh calculation fst-conv get-fh.elims*)
show *get-gs* (*add-empty-guards h*) = *get-gs x*
by (*metis add-empty-guards-def assms calculation get-gs.elims no-guard-remove*(*1*) *plus-comm snd-eqD*)
show *get-gu* (*add-empty-guards h*) = *get-gu x*
by (*metis add-empty-guards-def assms calculation get-gu.elims no-guard-remove*(*2*) *plus-comm snd-eqD*)
qed
ultimately show *?thesis* **by** *blast*
qed

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 EmptyFullGuards*
proof −
have (*s*, (*Map.empty*, *Some* (*pwrite*, {#}), (λ-. *Some* []))), (*s′*, (*Map.empty*, *Some* (*pwrite*, {#}), (λ-. *Some* []))) ⊨ *EmptyFullGuards*
by *simp*
then show *?thesis*
using *assms*(*1*) *assms*(*2*) *hyper-sat.simps*(*4*) *no-guard-add-empty-is-add* **by** *blast*
qed

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*
using *assms*(*1*) *assms*(*2*) *no-guard-add-empty-is-add*[*of a*] *no-guard-add-empty-is-add*[*of x*]
no-guard-then-smaller-same[*of x a b*] *plus-asso plus-comm*
by (*metis* (*no-types*, *lifting*))

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 (*rule semi-consistentI*)
show *all-guards* (*add-empty-guards h*)
by (*simp add*: *add-empty-guards-def all-guards-def*)
show *reachable* Γ (*view* Γ (*FractionalHeap.normalize* (*get-fh h*))) (*add-empty-guards h*)
proof (*rule reachableI*)
fix *sargs uargs*
assume *asm0*: *get-gs* (*add-empty-guards h*) = *Some* (*pwrite*, *sargs*) ∧ (∀ *k*. *get-gu* (*add-empty-guards h*) *k* = *Some* (*uargs k*))
then have *sargs* = {#} ∧ *uargs* = (λ*k*. [])
by (*metis add-empty-guards-def fst-conv get-gs.simps get-gu.simps option.sel snd-conv*)
then show *reachable-value* (*saction* Γ) (*uaction* Γ) (*view* Γ (*FractionalHeap.normalize* (*get-fh h*))) *sargs uargs*
(*view* Γ (*FractionalHeap.normalize* (*get-fh* (*add-empty-guards h*))))
by (*simp add*: *Self add-empty-guards-def*)
qed
qed

lemma *no-guards-remove-same*:
assumes *no-guard h*
shows *h* = *remove-guards* (*add-empty-guards h*)
by (*metis add-empty-guards-def addition-cancellative assms fst-conv get-fh.elims get-fh-remove-guards no-guard-add-empty-is-add no-guard-remove-guards*)

lemma *no-guards-remove*:
no-guard h ⟷ *h* = *remove-guards h*
by (*metis get-fh-remove-guards no-guard-remove-guards no-guards-remove-same remove-guards-def*)

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*
by (*simp add*: *add-sguard-to-no-guard-def*)

lemma *add-sguard-as-sum*:
assumes *no-guard h*
shows *Some* (*add-sguard-to-no-guard h* π *ms*) = *Some h* ⊕ *Some* (*Map.empty*, *Some* (π, *ms*), (λ-. *None*))
proof −
obtain *x* **where** *Some x* = *Some h* ⊕ *Some* (*Map.empty*, *Some* (π, *ms*), (λ-. *None*))

by (*simp add*: *assms no-guard-map-empty-compatible*)
moreover have $x = \textit{add-sguard-to-no-guard } h\ \pi\ ms$
proof (*rule heap-ext*)
show $\textit{get-fh } x = \textit{get-fh } (\textit{add-sguard-to-no-guard } h\ \pi\ ms)$
by (*metis add-fh-map-empty add-get-fh calculation fst-conv get-fh.elims get-fh-add-sguard*)
show $\textit{get-gs } x = \textit{get-gs } (\textit{add-sguard-to-no-guard } h\ \pi\ ms)$
by (*metis add-sguard-to-no-guard-def assms calculation get-gs.elims no-guard-def plus-comm snd-eqD sum-gs-one-none*)
show $\textit{get-gu } x = \textit{get-gu } (\textit{add-sguard-to-no-guard } h\ \pi\ ms)$
by (*metis add-sguard-to-no-guard-def assms calculation get-gu.simps no-guard-remove*(*2*) *plus-comm snd-conv*)
qed
ultimately show *?thesis* **by** *blast*
qed

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

lemma *get-fh-add-uguard*:
$\textit{get-fh } (\textit{add-uguard-to-no-guard } k\ h\ l) = \textit{get-fh } h$
by (*simp add*: *add-uguard-to-no-guard-def*)

lemma *prove-sum*:
assumes $a\ \#\#\ b$
and $\bigwedge x.\ \textit{Some } x = \textit{Some } a \oplus \textit{Some } b \Longrightarrow x = y$
shows $\textit{Some } y = \textit{Some } a \oplus \textit{Some } b$
using *assms*(*1*) *assms*(*2*) **by** *fastforce*

lemma *add-uguard-as-sum*:
assumes *no-guard h*
shows $\textit{Some } (\textit{add-uguard-to-no-guard } k\ h\ l) = \textit{Some } h \oplus \textit{Some } (\textit{Map.empty}, \textit{None}, (\lambda\text{-}. \textit{None})(k := \textit{Some } l))$
proof (*rule prove-sum*)
show $h\ \#\#\ (\textit{Map.empty}, \textit{None}, [k \mapsto l])$
by (*simp add*: *assms no-guard-map-empty-compatible*)
fix *x* **assume** *asm0*: $\textit{Some } x = \textit{Some } h \oplus \textit{Some } (\textit{Map.empty}, \textit{None}, [k \mapsto l])$
show $x = \textit{add-uguard-to-no-guard } k\ h\ l$
proof (*rule heap-ext*)
show $\textit{get-fh } x = \textit{get-fh } (\textit{add-uguard-to-no-guard } k\ h\ l)$
by (*metis add-fh-map-empty add-get-fh asm0 fst-conv get-fh.elims get-fh-add-uguard*)
show $\textit{get-gs } x = \textit{get-gs } (\textit{add-uguard-to-no-guard } k\ h\ l)$
by (*metis add-uguard-to-no-guard-def asm0 assms get-gs.elims no-guard-def plus-comm snd-eqD sum-gs-one-none*)
show $\textit{get-gu } x = \textit{get-gu } (\textit{add-uguard-to-no-guard } k\ h\ l)$
by (*metis add-uguard-to-no-guard-def asm0 assms get-gu.elims no-guard-remove*(*2*) *plus-comm snd-eqD*)
qed

qed

lemma *no-guard-and-no-heap*:
  assumes *Some h* = *Some p* ⊕ *Some g*
      and *no-guard p*
      and *get-fh g* = *Map.empty*
    shows *remove-guards h* = *p*
proof (*rule heap-ext*)
  show *get-fh* (*remove-guards h*) = *get-fh p*
  proof −
    have *get-fh* (*remove-guards h*) = *get-fh h*
      using *get-fh-remove-guards* **by** *blast*
    moreover have *get-fh h* = *add-fh* (*get-fh p*) (*get-fh g*)
      using *add-get-fh assms*(*1*) **by** *blast*
    ultimately show *?thesis*
      by (*metis assms*(*1*) *assms*(*3*) *ext get-fh.simps sum-second-none-get-fh*)
  qed
  show *get-gs* (*remove-guards h*) = *get-gs p*
    by (*metis assms*(*2*) *no-guard-def no-guard-remove-guards*)
  show *get-gu* (*remove-guards h*) = *get-gu p*
    by (*metis* ‹*get-fh* (*remove-guards h*) = *get-fh p*› *assms*(*2*) *get-fh-remove-guards no-guards-remove remove-guards-def*)
qed

lemma *decompose-guard-remove-easy*:
  Some h = *Some* (*remove-guards h*) ⊕ *Some* (*Map.empty*, *get-gs h*, *get-gu h*)
proof (*rule prove-sum*)
  show *remove-guards h* ## (*Map.empty*, *get-gs h*, *get-gu h*)
    by (*simp add*: *no-guard-map-empty-compatible no-guard-remove-guards*)
  fix *x* **assume** *asm0*: *Some x* = *Some* (*remove-guards h*) ⊕ *Some* (*Map.empty*, *get-gs h*, *get-gu h*)
  show *x* = *h*
  proof (*rule heap-ext*)
    show *get-fh x* = *get-fh h*
     by (*metis add-fh-map-empty add-get-fh asm0 fst-conv get-fh.elims get-fh-remove-guards*)
    show *get-gs x* = *get-gs h*
     by (*metis asm0 fst-conv get-gs.simps no-guard-remove*(*1*) *no-guard-remove-guards plus-comm snd-conv*)
    show *get-gu x* = *get-gu h*
      by (*metis asm0 get-gu.elims no-guard-remove*(*2*) *no-guard-remove-guards plus-comm snd-eqD*)
  qed
qed

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*
by (*metis all-guards-def assms*(*1*) *assms*(*2*) *assms*(*3*) *no-guard-def no-guard-remove*(*2*) *plus-comm sum-gs-one-none*)

lemma *all-guards-exists-uargs*:
assumes *all-guards x*
shows ∃ *uargs*. ∀ *k*. *get-gu x k* = *Some* (*uargs k*)
proof −
let *?uargs* = λ*k*. *the* (*get-gu x k*)
have ⋀*k*. *get-gu x k* = *Some* (*?uargs k*)
by (*metis all-guards-def assms option.collapse*)
then show *?thesis*
by *fastforce*
qed

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 (*cases* π = *pwrite*)
case *True*
then have *get-gs b* = *None*
using *add-gs.simps*(*2*)[*of* (π, *ms*)] *add-gs-cancellative add-gs-comm assms*(*1*) *assms*(*4*) *full-sguard-sum-same*
plus-extract(*2*)[*of x a b*]
by *metis*
moreover obtain *uargs* **where** ⋀*k*. *get-gu x k* = *Some* (*uargs k*)
using *all-guards-exists-uargs assms*(*2*) **by** *blast*
moreover have ⋀*k*. *get-gu b k* = *Some* (*uargs k*)
proof −
fix *k*
have *get-gu a k* = *None*
using *assms*(*3*) **by** *auto*
then show *get-gu b k* = *Some* (*uargs k*)
by (*metis* (*no-types*, *opaque-lifting*) *add-gu-def add-gu-single.simps*(*1*) *assms*(*1*) *calculation*(*2*) *plus-extract*(*3*))
qed
ultimately show *?thesis*
using *True* **by** *blast*
next
case *False*
then obtain π′ *msf* **where** *get-gs b* = *Some* (π′, *msf*)
by (*metis all-guards-def assms*(*1*) *assms*(*2*) *assms*(*4*) *fst-conv option.exhaust-sel*

option.sel prod.exhaust-sel sum-gs-one-none)
moreover obtain *v* **where** *get-gs x* = *Some* (*pwrite*, *v*)
by (*meson all-guards-def assms*(*2*))
ultimately have *pwrite* = *padd* π π'
by (*metis Pair-inject assms*(*1*) *assms*(*4*) *option.inject sum-gs-one-some*)
then show *?thesis*
by (*metis* (*mono-tags*, *opaque-lifting*) ‹*get-gs b* = *Some* (π', *msf*)› *add-gu-def add-gu-single.simps*(*1*) *all-guards-exists-uargs assms*(*1*) *assms*(*2*) *assms*(*3*) *plus-extract*(*3*))
qed

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*
apply (*simp add*: *denormalize-def no-guardI*)
using *full-ownershipI*[*of get-fh* (*denormalize H*)] *add-pwrite-option.elims denormalize-def fst-conv get-fh.elims option.distinct*(*1*) *option.sel* **apply** *metis*
proof −
show *normalize* (*get-fh* (*denormalize H*)) = *H*
proof (*rule ext*)
fix *l* **show** *normalize* (*get-fh* (*denormalize H*)) *l* = *H l*
by (*metis FractionalHeap.normalize-eq*(*1*) *FractionalHeap.normalize-eq*(*2*) *add-pwrite-option.elims denormalize-def fst-conv get-fh.elims*)
qed
show *full-ownership* (*get-fh h*) ∧ *no-guard h* ⟹ *denormalize* (*FractionalHeap.normalize* (*get-fh h*)) = *h*
proof −
assume *asm0*: *full-ownership* (*get-fh h*) ∧ *no-guard h*
show *denormalize* (*FractionalHeap.normalize* (*get-fh h*)) = *h*
proof (*rule heap-ext*)
show *get-fh* (*denormalize* (*FractionalHeap.normalize* (*get-fh h*))) = *get-fh h*
proof (*rule ext*)
fix *x* **show** *get-fh* (*denormalize* (*FractionalHeap.normalize* (*get-fh h*))) *x* = *get-fh h x*
proof (*cases get-fh h x*)
case *None*
then show *?thesis*
by (*metis FractionalHeap.normalize-eq*(*1*) *add-pwrite-option.simps*(*1*)

denormalize-def fst-conv get-fh.elims)
next
case (*Some p*)
then have *fst p* = *pwrite*
by (*meson asm0 full-ownership-def*)
then show *?thesis*
by (*metis FractionalHeap.normalize-eq*(*2*) *Some add-pwrite-option.simps*(*2*) *denormalize-def fst-conv get-fh.elims prod.collapse*)
qed
qed
show *get-gs* (*denormalize* (*FractionalHeap.normalize* (*get-fh h*))) = *get-gs h*
by (*metis asm0 denormalize-def fst-conv get-gs.elims no-guard-def snd-eqD*)
show *get-gu* (*denormalize* (*FractionalHeap.normalize* (*get-fh h*))) = *get-gu h*
by (*metis* ‹*get-fh* (*denormalize* (*FractionalHeap.normalize* (*get-fh h*))) = *get-fh h*› ‹*get-gs* (*denormalize* (*FractionalHeap.normalize* (*get-fh h*))) = *get-gs h*› *asm0 denormalize-def full-no-guard-same-normalize get-gu.simps no-guard-def snd-conv*)
qed
qed
assume *asm0*: *full-ownership* (*get-fh h*)
show *denormalize* (*FractionalHeap.normalize* (*get-fh h*)) = *remove-guards h*
proof (*rule heap-ext*)
show *get-fh* (*denormalize* (*FractionalHeap.normalize* (*get-fh h*))) = *get-fh* (*remove-guards h*)
proof (*rule ext*)
fix *x* **show** *get-fh* (*denormalize* (*FractionalHeap.normalize* (*get-fh h*))) *x* = *get-fh* (*remove-guards h*) *x*
proof (*cases get-fh h x*)
case *None*
then show *?thesis*
by (*metis FractionalHeap.normalize-eq*(*1*) *add-pwrite-option.simps*(*1*) *denormalize-def fst-eqD get-fh.elims get-fh-remove-guards*)
next
case (*Some p*)
then have *fst p* = *pwrite*
by (*meson asm0 full-ownership-def*)
then show *?thesis*
by (*metis FractionalHeap.normalize-eq*(*2*) *Some add-pwrite-option.simps*(*2*) *denormalize-def fst-conv get-fh.elims get-fh-remove-guards prod.collapse*)
qed
qed
show *get-gs* (*denormalize* (*FractionalHeap.normalize* (*get-fh h*))) = *get-gs* (*remove-guards h*)
by (*simp add*: *denormalize-def remove-guards-def*)
show *get-gu* (*denormalize* (*FractionalHeap.normalize* (*get-fh h*))) = *get-gu* (*remove-guards h*)
by (*metis* ‹*get-fh* (*denormalize* (*FractionalHeap.normalize* (*get-fh h*))) = *get-fh* (*remove-guards h*)› ‹*get-gs* (*denormalize* (*FractionalHeap.normalize* (*get-fh h*))) = *get-gs* (*remove-guards h*)› *asm0 denormalize-def full-no-guard-same-normalize get-fh-remove-guards get-gu.simps no-guard-def no-guard-remove-guards snd-conv*)

qed
qed

lemma *no-guard-then-sat-star-uguard*:
assumes *no-guard* $h \wedge$ *no-guard* h'
and $(s, h), (s', h') \models Q$
shows $(s,$ *add-uguard-to-no-guard* $k\ h\ (e\ s)), (s',$ *add-uguard-to-no-guard* $k\ h'$ $(e\ s')) \models$ *Star* Q (*UniqueGuard* $k\ e$)
proof −
obtain *Some* (*add-uguard-to-no-guard* $k\ h\ (e\ s)$) = *Some* $h \oplus$ *Some* (*Map.empty*, *None*, $[k \mapsto e\ s]$)
Some (*add-uguard-to-no-guard* $k\ h'\ (e\ s')$) = *Some* $h' \oplus$ *Some* (*Map.empty*, *None*, $[k \mapsto e\ s']$)
by (*simp add*: *add-uguard-as-sum assms*(*1*))
moreover have $(s,$ (*Map.empty*, *None*, $[k \mapsto e\ s]$)), $(s',$ (*Map.empty*, *None*, $[k \mapsto e\ s']$)) $\models$ *UniqueGuard* $k\ e$
by *simp*
ultimately show *?thesis* **using** *assms*(*2*) **by** *fastforce*
qed

lemma *no-guard-then-sat-star*:
assumes *no-guard* $h \wedge$ *no-guard* h'
and $(s, h), (s', h') \models Q$
shows $(s,$ *add-sguard-to-no-guard* $h\ \pi\ (ms\ s)), (s',$ *add-sguard-to-no-guard* $h'\ \pi$ $(ms\ s')) \models$ *Star* Q (*SharedGuard* π *ms*)
proof −
obtain *Some* (*add-sguard-to-no-guard* $h\ \pi\ (ms\ s)$) = *Some* $h \oplus$ *Some* (*Map.empty*, *Some* $(\pi, ms\ s)$, (λ-. *None*))
Some (*add-sguard-to-no-guard* $h'\ \pi\ (ms\ s')$) = *Some* $h' \oplus$ *Some* (*Map.empty*, *Some* $(\pi, ms\ s')$, (λ-. *None*))
using *add-sguard-as-sum assms*(*1*) **by** *blast*
moreover have $(s,$ (*Map.empty*, *Some* $(\pi, ms\ s)$, (λ-. *None*))), $(s',$ (*Map.empty*, *Some* $(\pi, ms\ s')$, (λ-. *None*))) $\models$ *SharedGuard* π *ms*
by *simp*
ultimately show *?thesis* **using** *assms*(*2*) **by** *fastforce*
qed

end

4.3 Safety and Hoare Triples

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

theory *Safety*
imports *Guards*
begin

4.3.1 Preliminaries

definition *sat-inv* :: *store* ⇒ ($'i$, $'a$) *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
  by (*simp add*: *assms*(*1*) *assms*(*2*) *sat-inv-def*)

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*
  using *assms upper-fvs-def* **by** *blast*

lemma *pair-sat-comm*:
  assumes *pair-sat S S' A*
  shows *pair-sat S' S A*
proof (*rule pair-satI*)
  fix *s h s' h'* **assume** (*s*, *h*) ∈ *S'* ∧ (*s'*, *h'*) ∈ *S*
  then show (*s*, *h*), (*s'*, *h'*) ⊨ *A*
    using *assms pair-sat-def sat-comm* **by** *blast*
qed

lemma *in-upperize*:
  (*s'*, *h*) ∈ *upperize S vars* ⟷ (∃ *s*. (*s*, *h*) ∈ *S* ∧ *agrees vars s s'*) (**is** *?A* ⟷ *?B*)
proof
  show *?A* ⟹ *?B*
    by (*simp add*: *upperize-def*)
  show *?B* ⟹ *?A*
    using *upperize-def* **by** *fastforce*
qed

lemma *upper-fvs-upperize*:

upper-fvs (*upperize S vars*) *vars*
proof (*rule upper-fvsI*)
fix *s s′ h*
assume (*s*, *h*) ∈ *upperize S vars* ∧ *agrees vars s s′*
then obtain *s′′* **where** (*s′′*, *h*) ∈ *S* ∧ *agrees vars s′′ s*
by (*meson in-upperize*)
then have *agrees vars s′′ s′*
using ‹(*s*, *h*) ∈ *upperize S vars* ∧ *agrees vars s s′*› *agrees-def*[*of vars s s′*]
agrees-def[*of vars s′′ s*] *agrees-def*[*of vars s′′ s′*]
by *simp*
then show (*s′*, *h*) ∈ *upperize S vars*
using ‹(*s′′*, *h*) ∈ *S* ∧ *agrees vars s′′ s*› *upperize-def* **by** *fastforce*
qed

lemma *upperize-larger*:
S ⊆ *upperize S vars*
proof
fix *x* **assume** *x* ∈ *S*
moreover have *agrees vars* (*fst x*) (*fst x*)
using *agrees-def* **by** *blast*
ultimately show *x* ∈ *upperize S vars*
by (*metis* (*mono-tags*, *lifting*) *CollectI upperize-def*)
qed

lemma *pair-sat-upperize*:
assumes *pair-sat S S′ A*
shows *pair-sat* (*upperize S* (*fvA A*)) *S′ A*
proof (*rule pair-satI*)
fix *s h s′ h′*
assume *asm0*: (*s*, *h*) ∈ *upperize S* (*fvA A*) ∧ (*s′*, *h′*) ∈ *S′*
then obtain *s′′* **where** *agrees* (*fvA A*) *s s′′* (*s′′*, *h*) ∈ *S*
using *agrees-def*[*of fvA A s s′*] *in-upperize*[*of s h S fvA A*]
by (*metis agrees-def*)
then show (*s*, *h*), (*s′*, *h′*) ⊨ *A*
using *agrees-same asm0 assms pair-sat-def* **by** *blast*
qed

lemma *in-close-var*:
(*s′*, *h*) ∈ *close-var S x* ⟷ (∃ *s v*. (*s*, *h*) ∈ *S* ∧ *s′* = *s*(*x* := *v*)) (**is** *?A* ⟷ *?B*)
proof
show *?A* ⟹ *?B*
using *close-var-def*[*of S x*] *mem-Collect-eq prod.inject surjective-pairing*
by *auto*
show *?B* ⟹ *?A*
using *close-var-def* **by** *fastforce*
qed

lemma *pair-sat-close-var*:
assumes *x* ∉ *fvA A*

and *pair-sat* S S' A
shows *pair-sat* (*close-var* S x) S' A
proof (*rule pair-satI*)
fix s h s' h'
assume $(s, h) \in$ *close-var* S x $\wedge$ $(s', h') \in S'$
then show $(s, h), (s', h') \models A$
by (*metis* (*no-types*, *lifting*) *agrees-same agrees-update assms in-close-var pair-sat-def*)
qed

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
using *assms pair-sat-close-var pair-sat-comm* **by** *blast*

lemma *close-var-subset*:
$S \subseteq$ *close-var* S x
proof
fix y **assume** $y \in S$
then have *fst* y = (*fst* y)(x := (*fst* y x))
by *simp*
then show $y \in$ *close-var* S x
by (*metis* ‹$y \in S$› *in-close-var prod.exhaust-sel*)
qed

lemma *upper-fvs-close-vars*:
upper-fvs (*close-var* S x) $(- \{x\})$
proof (*rule upper-fvsI*)
fix s s' h
assume $(s, h) \in$ *close-var* S x $\wedge$ *agrees* $(- \{x\})$ s s'
have $s(x := s'\ x) = s'$
proof (*rule ext*)
fix y **show** $(s(x := s'\ x))\ y = s'\ y$
by (*metis* (*mono-tags*, *lifting*) *ComplI* ‹$(s, h) \in$ *close-var* S x $\wedge$ *agrees* $(- \{x\})$ s s'› *agrees-def fun-upd-apply singleton-iff*)
qed
then show $(s', h) \in$ *close-var* S x
by (*metis* ‹$(s, h) \in$ *close-var* S x $\wedge$ *agrees* $(- \{x\})$ s s'› *fun-upd-upd in-close-var*)
qed

lemma *sat-inv-agrees*:
assumes *sat-inv* s *hj* Γ
and *agrees* (*fvA* (*invariant* Γ)) s s'
shows *sat-inv* s' *hj* Γ
by (*meson agrees-same assms sat-comm sat-inv-def*)

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

using *aborts-agrees assms fst-conv snd-eqD*
by (*metis* (*mono-tags*, *lifting*) *agrees-def*)

lemma *view-function-of-invE*:
assumes *view-function-of-inv* Γ
and *sat-inv s h* Γ
and $(h' :: ('i, 'a)\ heap) \succeq h$
shows *view* Γ (*normalize* (*get-fh h*)) = *view* Γ (*normalize* (*get-fh h'*))
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *sat-inv-def view-function-of-inv-def* **by** *blast*

4.3.2 Safety

fun *no-abort* :: ('*i*, '*a*, *nat*) *cont* $\Rightarrow$ *cmd* $\Rightarrow$ *store* $\Rightarrow$ ('*i*, '*a*) *heap* $\Rightarrow$ *bool* **where**
no-abort None C s h $\longleftrightarrow$ ($\forall$ *hf H*. *Some H* = *Some h* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H*
$\longrightarrow \neg$ *aborts C* (*s*, *normalize* (*get-fh H*)))
| *no-abort* (*Some* Γ) *C s h* $\longleftrightarrow$ ($\forall$ *hf H hj v0*. *Some H* = *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$
semi-consistent Γ *v0 H* $\wedge$ *sat-inv s hj* Γ
$\longrightarrow \neg$ *aborts C* (*s*, *normalize* (*get-fh H*)))

lemma *no-abortI*:
assumes $\bigwedge$(*hf* :: ('*i*, '*a*) *heap*) (*H* :: ('*i*, '*a*) *heap*). *Some H* = *Some h* $\oplus$ *Some hf* $\wedge$ Δ = *None* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H* $\Longrightarrow \neg$ *aborts C* (*s*, *normalize* (*get-fh H*))
and $\bigwedge$*H hf hj v0* Γ. Δ = *Some* Γ $\wedge$ *Some H* = *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s hj* Γ
$\Longrightarrow \neg$ *aborts C* (*s*, *normalize* (*get-fh H*))
shows *no-abort* Δ *C s* (*h* :: ('*i*, '*a*) *heap*)
apply (*cases* Δ)
using *assms*(*1*) *no-abort.simps*(*1*) **apply** *blast*
using *assms*(*2*) *no-abort.simps*(*2*) **by** *blast*

lemma *no-abortSomeI*:
assumes $\bigwedge$*H hf hj v0*. *Some H* = *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s hj* Γ
$\Longrightarrow \neg$ *aborts C* (*s*, *normalize* (*get-fh H*))
shows *no-abort* (*Some* Γ) *C s* (*h* :: ('*i*, '*a*) *heap*)
using *assms no-abort.simps*(*2*) **by** *blast*

lemma *no-abortNoneI*:
assumes $\bigwedge$(*hf* :: ('*i*, '*a*) *heap*) (*H* :: ('*i*, '*a*) *heap*). *Some H* = *Some h* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H* $\Longrightarrow \neg$ *aborts C* (*s*, *normalize* (*get-fh H*))
shows *no-abort* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C s* (*h* :: ('*i*, '*a*) *heap*)
using *assms no-abort.simps*(*1*) **by** *blast*

lemma *no-abortE*:
assumes *no-abort* Δ *C s h*

shows $Some\ H = Some\ h \oplus Some\ hf \Longrightarrow \Delta = None \Longrightarrow full\text{-}ownership\ (get\text{-}fh\ H) \Longrightarrow no\text{-}guard\ H \Longrightarrow \neg\ aborts\ C\ (s,\ normalize\ (get\text{-}fh\ H))$
and $\Delta = Some\ \Gamma \Longrightarrow Some\ H = Some\ h \oplus Some\ hj \oplus Some\ hf \Longrightarrow sat\text{-}inv\ s\ hj\ \Gamma \Longrightarrow full\text{-}ownership\ (get\text{-}fh\ H) \Longrightarrow semi\text{-}consistent\ \Gamma\ v0\ H$
$\Longrightarrow \neg\ aborts\ C\ (s,\ normalize\ (get\text{-}fh\ H))$
using *assms no-abort.simps(1)* **apply** *blast*
by (*metis assms no-abort.simps(2)*)

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

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

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

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

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

lemma *safeSomeI*:
assumes $C = Cskip \Longrightarrow (s,\ h) \in S$
and $no\text{-}abort\ (Some\ \Gamma)\ C\ s\ h$

and $accesses\ C\ s \subseteq dom\ (fst\ h) \wedge writes\ C\ s \subseteq fpdom\ (fst\ h)$
and $\bigwedge H\ hf\ C'\ s'\ h'\ hj\ v0.\ Some\ H = Some\ h \oplus Some\ hj \oplus Some\ hf \wedge full\text{-}ownership\ (get\text{-}fh\ H)$
$\wedge\ semi\text{-}consistent\ \Gamma\ v0\ H \wedge sat\text{-}inv\ s\ hj\ \Gamma \wedge red\ C\ (s,\ normalize\ (get\text{-}fh\ H))\ C'\ (s',\ h')$
$\Longrightarrow (\exists\, h''\ H'\ hj'.\ full\text{-}ownership\ (get\text{-}fh\ H') \wedge semi\text{-}consistent\ \Gamma\ v0\ H' \wedge sat\text{-}inv\ s'\ hj'\ \Gamma$
$\wedge\ h' = normalize\ (get\text{-}fh\ H') \wedge Some\ H' = Some\ h'' \oplus Some\ hj' \oplus Some\ hf \wedge safe\ n\ (Some\ \Gamma)\ C'\ (s',\ h'')\ S)$
shows $safe\ (Suc\ n)\ (Some\ \Gamma)\ C\ (s,\ h :: ('i,\ 'a)\ heap)\ S$
using *assms* **by** *auto*

lemma *safeI*:
fixes $\Delta :: ('i,\ 'a,\ nat)\ cont$
assumes $C = Cskip \Longrightarrow (s,\ h) \in S$
and $no\text{-}abort\ \Delta\ C\ s\ h$
and $accesses\ C\ s \subseteq dom\ (fst\ h) \wedge writes\ C\ s \subseteq fpdom\ (fst\ h)$
and $\bigwedge H\ hf\ C'\ s'\ h'.\ \Delta = None \Longrightarrow Some\ H = Some\ h \oplus Some\ hf \wedge full\text{-}ownership\ (get\text{-}fh\ H) \wedge no\text{-}guard\ H \wedge red\ C\ (s,\ normalize\ (get\text{-}fh\ H))\ C'\ (s',\ h')$
$\Longrightarrow (\exists\, h''\ H'.\ full\text{-}ownership\ (get\text{-}fh\ H') \wedge no\text{-}guard\ H' \wedge h' = normalize\ (get\text{-}fh\ H') \wedge Some\ H' = Some\ h'' \oplus Some\ hf \wedge safe\ n\ (None :: ('i,\ 'a,\ nat)\ cont)\ C'\ (s',\ h'')\ S)$
and $\bigwedge H\ hf\ C'\ s'\ h'\ hj\ v0\ \Gamma.\ \Delta = Some\ \Gamma \Longrightarrow Some\ H = Some\ h \oplus Some\ hj \oplus Some\ hf \wedge full\text{-}ownership\ (get\text{-}fh\ H)$
$\wedge\ semi\text{-}consistent\ \Gamma\ v0\ H \wedge sat\text{-}inv\ s\ hj\ \Gamma \wedge red\ C\ (s,\ normalize\ (get\text{-}fh\ H))\ C'\ (s',\ h')$
$\Longrightarrow (\exists\, h''\ H'\ hj'.\ full\text{-}ownership\ (get\text{-}fh\ H') \wedge semi\text{-}consistent\ \Gamma\ v0\ H' \wedge sat\text{-}inv\ s'\ hj'\ \Gamma$
$\wedge\ h' = normalize\ (get\text{-}fh\ H') \wedge Some\ H' = Some\ h'' \oplus Some\ hj' \oplus Some\ hf \wedge safe\ n\ (Some\ \Gamma)\ C'\ (s',\ h'')\ S)$
shows $safe\ (Suc\ n)\ \Delta\ C\ (s,\ h :: ('i,\ 'a)\ heap)\ S$
proof (*cases* Δ)
case *None*
then show *?thesis*
using *assms* **by** *auto*
next
case (*Some* Γ)
then show *?thesis* **using** *safeSomeI assms*
by *simp*
qed

lemma *safeSomeAltI*:
assumes $C = Cskip \Longrightarrow (s,\ h) \in S$
and $\bigwedge H\ hf\ hj\ v0.\ Some\ H = Some\ h \oplus Some\ hj \oplus Some\ hf \wedge full\text{-}ownership\ (get\text{-}fh\ H) \wedge semi\text{-}consistent\ \Gamma\ v0\ H \wedge sat\text{-}inv\ s\ hj\ \Gamma$
$\Longrightarrow \neg\ aborts\ C\ (s,\ normalize\ (get\text{-}fh\ H))$
and $\bigwedge H\ hf\ C'\ s'\ h'\ hj\ v0.\ Some\ H = Some\ h \oplus Some\ hj \oplus Some\ hf \wedge$

full-ownership (*get-fh* H)
$\wedge$ *semi-consistent* Γ *v0* H $\wedge$ *sat-inv* s *hj* $\Gamma \Longrightarrow$ *red* C (s, *normalize* (*get-fh* H)) C' (s', h')
$\Longrightarrow$ ($\exists\, h''\ H'\ hj'$. *full-ownership* (*get-fh* H') $\wedge$ *semi-consistent* Γ *v0* H' $\wedge$ *sat-inv* s' hj' Γ
$\wedge$ h' = *normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some* hj' $\oplus$ *Some* *hf* $\wedge$ *safe* n (*Some* Γ) C' (s', h'') S)
and *accesses* C $s \subseteq$ *dom* (*fst* h) $\wedge$ *writes* C $s \subseteq$ *fpdom* (*fst* h)
shows *safe* (*Suc* n) (*Some* Γ) C (s, h :: ('*i*, '*a*) *heap*) S
using *assms*(*1*)
proof (*rule safeSomeI*)
show *no-abort* (*Some* Γ) C s h **using** *assms*(*2*) *no-abortSomeI* **by** *blast*
show $\bigwedge H$ *hf* C' s' h' *hj* *v0*.
Some H = *Some* h $\oplus$ *Some* *hj* $\oplus$ *Some* *hf* $\wedge$ *full-ownership* (*get-fh* H) $\wedge$ *semi-consistent* Γ *v0* H $\wedge$ *sat-inv* s *hj* Γ $\wedge$ *red* C (s, *FractionalHeap.normalize* (*get-fh* H)) C' (s', h') $\Longrightarrow$
($\exists\, h''\ H'\ hj'$.
full-ownership (*get-fh* H') $\wedge$
semi-consistent Γ *v0* H' $\wedge$ *sat-inv* s' hj' Γ $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some* hj' $\oplus$ *Some* *hf* $\wedge$ *safe* n (*Some* Γ) C' (s', h'') S)
using *assms*(*3*) **by** *blast*
qed (*auto simp add*: *assms*)

lemma *safeAccessesE*:
assumes *safe* (*Suc* n) Δ C σ S
shows *accesses* C (*fst* σ) $\subseteq$ *dom* (*fst* (*snd* σ)) $\wedge$ *writes* C (*fst* σ) $\subseteq$ *fpdom* (*fst* (*snd* σ))
apply (*cases* Δ)
using *assms safe.simps*(*2*)[*of n C fst* σ *snd* σ *S*] *safe.simps*(*3*)[*of n - C fst* σ *snd* σ *S*] **by** *simp-all*

lemma *safeSomeE*:
assumes *safe* (*Suc* n) (*Some* Γ) C (s, h :: ('*i*, '*a*) *heap*) S
shows C = *Cskip* $\Longrightarrow$ (s, h) $\in S$
and *no-abort* (*Some* Γ) C s h
and *Some* H = *Some* h $\oplus$ *Some* *hj* $\oplus$ *Some* *hf* $\Longrightarrow$ *full-ownership* (*get-fh* H)
$\Longrightarrow$ *semi-consistent* Γ *v0* H $\Longrightarrow$ *sat-inv* s *hj* $\Gamma \Longrightarrow$ *red* C (s, *normalize* (*get-fh* H)) C' (s', h')
$\Longrightarrow$ ($\exists\, h''\ H'\ hj'$. *full-ownership* (*get-fh* H') $\wedge$ *semi-consistent* Γ *v0* H' $\wedge$ *sat-inv* s' hj' Γ
$\wedge$ h' = *normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some* hj' $\oplus$ *Some* *hf* $\wedge$ *safe* n (*Some* Γ) C' (s', h'') S)
using *assms safe.simps*(*3*)[*of n* Γ *C s h S*] **by** *blast+*

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-abort* (*None* :: ('*i*, '*a*, *nat*) *cont*) C s h

and $Some\ H = Some\ h \oplus Some\ hf \Longrightarrow full\text{-}ownership\ (get\text{-}fh\ H) \Longrightarrow no\text{-}guard\ H \Longrightarrow red\ C\ (s,\ normalize\ (get\text{-}fh\ H))\ C'\ (s',\ h')$
$\Longrightarrow (\exists\, h''\ H'.\ full\text{-}ownership\ (get\text{-}fh\ H') \wedge no\text{-}guard\ H' \wedge h' = normalize\ (get\text{-}fh\ H') \wedge Some\ H' = Some\ h'' \oplus Some\ hf \wedge safe\ n\ (None :: ('i,\ 'a,\ nat)\ cont)\ C'\ (s',\ h'')\ S)$
using *assms safe.simps(2)[of n C s h S]* **by** *blast+*

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 $Some\ H = Some\ h \oplus Some\ hf \Longrightarrow full\text{-}ownership\ (get\text{-}fh\ H) \Longrightarrow no\text{-}guard\ H \Longrightarrow red\ C\ (s,\ normalize\ (get\text{-}fh\ H))\ C'\ (s',\ h')$
$\Longrightarrow (\exists\, h''\ H'.\ full\text{-}ownership\ (get\text{-}fh\ H') \wedge no\text{-}guard\ H' \wedge h' = normalize\ (get\text{-}fh\ H') \wedge Some\ H' = Some\ h'' \oplus Some\ hf \wedge safe\ n\ no\text{-}cont\ C'\ (s',\ h'')\ S)$
using *assms safe.simps(2)[of n C s h S]* **by** *blast+*

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 (*rule no-abortI*)
show $\bigwedge hf\ H.\ Some\ H = Some\ h' \oplus Some\ hf \wedge \Gamma = None \wedge full\text{-}ownership\ (get\text{-}fh\ H) \wedge no\text{-}guard\ H \Longrightarrow \neg\ aborts\ C\ (s,\ FractionalHeap.normalize\ (get\text{-}fh\ H))$
using *assms(1) assms(2) larger-def larger-trans no-abort.simps(1)* **by** *blast*
show $\bigwedge H\ hf\ hj\ v0\ \Gamma'.$
$\Gamma = Some\ \Gamma' \wedge Some\ H = Some\ h' \oplus Some\ hj \oplus Some\ hf \wedge full\text{-}ownership\ (get\text{-}fh\ H) \wedge semi\text{-}consistent\ \Gamma'\ v0\ H \wedge sat\text{-}inv\ s\ hj\ \Gamma' \Longrightarrow$
$\neg\ aborts\ C\ (s,\ FractionalHeap.normalize\ (get\text{-}fh\ H))$
proof −
fix $H\ hf\ hj\ v0\ \Gamma'$
assume *asm0*: $\Gamma = Some\ \Gamma' \wedge Some\ H = Some\ h' \oplus Some\ hj \oplus Some\ hf \wedge full\text{-}ownership\ (get\text{-}fh\ H) \wedge semi\text{-}consistent\ \Gamma'\ v0\ H \wedge sat\text{-}inv\ s\ hj\ \Gamma'$
moreover obtain r **where** $Some\ h' = Some\ h \oplus Some\ r$
using *assms(1) larger-def* **by** *blast*
then obtain hf' **where** $Some\ hf' = Some\ hf \oplus Some\ r$
by (*metis (no-types, opaque-lifting) calculation not-None-eq plus.simps(1) plus-asso plus-comm*)
then have $Some\ H = Some\ h \oplus Some\ hj \oplus Some\ hf'$
by (*metis (no-types, opaque-lifting)* ‹$Some\ h' = Some\ h \oplus Some\ r$› *calculation plus-asso plus-comm*)
then show $\neg\ aborts\ C\ (s,\ FractionalHeap.normalize\ (get\text{-}fh\ H))$
using *assms(2) calculation no-abortE(2)* **by** *blast*
qed
qed

lemma *safe-larger-set-aux*:
fixes Δ :: ($'i$, $'a$, *nat*) *cont*
assumes *safe n* Δ *C* (*s*, *h*) *S*
and $S \subseteq S'$
shows *safe n* Δ *C* (*s*, *h*) S'
using *assms*
proof (*induct n arbitrary*: *s h C*)
case (*Suc n*)
show *?case*
proof (*rule safeI*)
show $C = Cskip \Longrightarrow (s, h) \in S'$
by (*metis* (*no-types*, *opaque-lifting*) *Suc.prems*(*1*) *assms*(*2*) *not-Some-eq safeNoneE-bis*(*1*) *safeSomeE*(*1*) *subset-iff*)
show *no-abort* Δ *C s h*
apply (*cases* Δ)
using *Suc.prems*(*1*) *safeNoneE-bis*(*2*) **apply** *blast*
using *Suc.prems*(*1*) *safeSomeE*(*2*) **by** *blast*

show $\bigwedge H\ hf\ C'\ s'\ h'.$
Δ = *None* $\Longrightarrow$
Some H = *Some h* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H* $\wedge$ *red C* (*s*, *FractionalHeap.normalize* (*get-fh H*)) C' (s', h') $\Longrightarrow$
$\exists h''\ H'$. *full-ownership* (*get-fh* H') $\wedge$ *no-guard* H' $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some hf* $\wedge$ *safe n* (*None* :: ($'i$, $'a$, *nat*) *cont*) C' (s', h'') S'
using *Suc.hyps Suc.prems*(*1*) *assms*(*2*) *safeNoneE*(*3*)[*of n C s h*] **by** *blast*

show $\bigwedge H\ hf\ C'\ s'\ h'\ hj\ v0\ \Gamma.$
Δ = *Some* Γ $\Longrightarrow$
Some H = *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$
full-ownership (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s hj* Γ $\wedge$ *red C* (*s*, *FractionalHeap.normalize* (*get-fh H*)) C' (s', h') $\Longrightarrow$
$\exists h''\ H'\ hj'$.
full-ownership (*get-fh* H') $\wedge$
semi-consistent Γ *v0* H' $\wedge$ *sat-inv* s' hj' Γ $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some* hj' $\oplus$ *Some hf* $\wedge$ *safe n* (*Some* Γ) C' (s', h'') S'
proof –
fix *H hf* C' s' h' *hj v0* Γ
assume *asm0*: Δ = *Some* Γ *Some H* = *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$
full-ownership (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s hj* Γ $\wedge$ *red C* (*s*, *FractionalHeap.normalize* (*get-fh H*)) C' (s', h')
then show $\exists h''\ H'\ hj'$. *full-ownership* (*get-fh* H') $\wedge$ *semi-consistent* Γ *v0* H' $\wedge$ *sat-inv* s' hj' Γ
$\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some* hj' $\oplus$ *Some hf* $\wedge$ *safe n* (*Some* Γ) C' (s', h'') S'
using *safeSomeE*(*3*)[*of n* Γ *C s h S*] *Suc.hyps Suc.prems*(*1*) *assms*(*2*) **by** *blast*

qed
show *accesses* C $s \subseteq$ *dom* (*fst* h) $\wedge$ *writes* C $s \subseteq$ *fpdom* (*fst* h)
by (*metis Suc.prems*(*1*) *fst-conv safeAccessesE snd-conv*)
qed
qed (*simp*)

lemma *safe-larger-set*:
assumes *safe* n Δ C σ S
and $S \subseteq S'$
shows *safe* n Δ C σ S'
using *assms safe-larger-set-aux*[*of* n Δ C *fst* σ *snd* σ S S']
by *auto*

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
using *assms*
proof (*induct* n *arbitrary*: s h C m)
case (*Suc* n)
show *?case*
proof (*cases* m)
case (*Suc* k)
then have $k \leq n$
using *Suc.prems*(*1*) **by** *fastforce*
moreover have *safe* (*Suc* k) Δ C (s, h) S
proof (*rule safeI*)
show C = *Cskip* $\Longrightarrow$ (s, h) $\in S$
using *Suc.prems*(*2*) *safe.elims*(*2*) **by** *blast*
show *no-abort* Δ C s h
apply (*cases* Δ)
using *Suc.prems*(*2*) *safeNoneE*(*2*) **apply** *blast*
using *Suc.prems*(*2*) *safeSomeE*(*2*) **by** *blast*
show $\bigwedge H$ *hf* C' s' h'.
Δ = *None* $\Longrightarrow$
Some H = *Some* h $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh* H) $\wedge$ *no-guard* H $\wedge$ *red* C (s, *FractionalHeap.normalize* (*get-fh* H)) C' (s', h') $\Longrightarrow$
$\exists h''$ H'. *full-ownership* (*get-fh* H') $\wedge$ *no-guard* H' $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some hf* $\wedge$ *safe* k (*None* :: ($'i$, $'a$, *nat*) *cont*) C' (s', h'') S
proof −
fix H *hf* C' s' h'
assume *asm0*: Δ = *None Some* H = *Some* h $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh* H) $\wedge$ *no-guard* H $\wedge$ *red* C (s, *FractionalHeap.normalize* (*get-fh* H)) C' (s', h')
then obtain h'' H' **where** *full-ownership* (*get-fh* H') $\wedge$ *no-guard* H' $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some hf* $\wedge$ *safe* n (*None* :: ($'i$, $'a$, *nat*) *cont*) C' (s', h'') S

using *Suc.prems(2) safeNoneE(3)* **by** *blast*
then show $\exists h'' H'$. *full-ownership (get-fh H')* $\wedge$ *no-guard H'* $\wedge$ $h' =$ *FractionalHeap.normalize (get-fh H')* $\wedge$ *Some H'* $=$ *Some h''* $\oplus$ *Some hf* $\wedge$ *safe k (None :: ('i, 'a, nat) cont) C' (s', h'') S*
using *Suc.hyps asm0(1) calculation* **by** *blast*
qed
show *accesses C s* $\subseteq$ *dom (fst h)* $\wedge$ *writes C s* $\subseteq$ *fpdom (fst h)*
by *(metis Suc.prems(2) fst-eqD safeAccessesE snd-eqD)*
fix *H hf C' s' h' hj v0* Γ
assume *asm0*: $\Delta =$ *Some* Γ *Some H* $=$ *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$ *full-ownership (get-fh H)* $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s hj* Γ $\wedge$ *red C (s, FractionalHeap.normalize (get-fh H)) C' (s', h')*
then show $\exists h'' H' hj'$.
full-ownership (get-fh H') $\wedge$
semi-consistent Γ *v0 H'* $\wedge$ *sat-inv s' hj'* Γ $\wedge$ $h' =$ *FractionalHeap.normalize (get-fh H')* $\wedge$ *Some H'* $=$ *Some h''* $\oplus$ *Some hj'* $\oplus$ *Some hf* $\wedge$ *safe k (Some* Γ*) C' (s', h'') S*
using *Suc.prems(2) safeSomeE(3)[of n* Γ *C s h S H hj hf v0 C' s' h'] Suc.hyps*
using *calculation* **by** *blast*
qed
ultimately show *?thesis*
using *Suc* **by** *auto*
qed *(simp)*
qed *(simp)*

lemma *safe-smaller*:
assumes $m \leq n$
and *safe n* Δ *C* σ *S*
shows *safe m* Δ *C* σ *S*
by *(metis assms(1) assms(2) safe-smaller-aux surj-pair)*

If it is safe to execute n steps of C in the state (s0, h), then it is also safe to execute it in the state (s1, h), provided that s0 and s1 agree on the values of variables that are free in C, the invariant, and the postcondition.

lemma *safe-free-vars-aux*:
fixes Δ *:: ('i, 'a, nat) cont*
assumes *safe n* Δ *C (s0, h) S*
and *agrees (fvC C* $\cup$ *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*
using *assms*
proof *(induct n arbitrary: s0 h s1 C)*
case *(Suc n)*
show *?case*
proof *(rule safeI)*
show $C =$ *Cskip* $\Longrightarrow$ *(s1, h)* $\in$ *S*
by *(metis Suc.prems(1) Suc.prems(2) agrees-union assms(3) not-Some-eq*

safeNoneE-bis(*1*) *safeSomeE*(*1*) *upper-fvs-def*)
show *no-abort* Δ *C s1 h*
proof (*rule no-abortI*)
show $\bigwedge$*hf H*. *Some H* = *Some h* $\oplus$ *Some hf* $\wedge$ Δ = *None* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H* $\Longrightarrow$ $\neg$ *aborts C* (*s1*, *FractionalHeap.normalize* (*get-fh H*))
using *Suc.prems*(*1*) *Suc.prems*(*2*) *abort-iff-fvC agrees-union no-abortE*(*1*) *safeNoneE*(*2*) **by** *blast*
show $\bigwedge$*H hf hj v0* Γ. Δ = *Some* Γ $\wedge$ *Some H* = *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s1 hj* Γ $\Longrightarrow$
$\neg$ *aborts C* (*s1*, *FractionalHeap.normalize* (*get-fh H*))
proof −
fix *H hf hj v0* Γ
assume *asm0*: Δ = *Some* Γ $\wedge$ *Some H* = *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s1 hj* Γ
then have *sat-inv s0 hj* Γ
using *Suc.prems*(*4*) *agrees-def sat-inv-agrees*
by (*metis* (*mono-tags*, *opaque-lifting*))
then have $\neg$ *aborts C* (*s0*, *FractionalHeap.normalize* (*get-fh H*))
using *Suc.prems*(*1*) *asm0 no-abort.simps*(*2*) *safeSomeE*(*2*) **by** *blast*
then show $\neg$ *aborts C* (*s1*, *FractionalHeap.normalize* (*get-fh H*))
using *Suc.prems*(*2*) *abort-iff-fvC agrees-union* **by** *blast*
qed
qed
show $\bigwedge$*H hf C' s1' h'*.
Δ = *None* $\Longrightarrow$
Some H = *Some h* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H* $\wedge$ *red C* (*s1*, *FractionalHeap.normalize* (*get-fh H*)) *C'* (*s1'*, *h'*) $\Longrightarrow$
$\exists$*h'' H'*. *full-ownership* (*get-fh H'*) $\wedge$ *no-guard H'* $\wedge$ *h'* = *FractionalHeap.normalize* (*get-fh H'*) $\wedge$ *Some H'* = *Some h''* $\oplus$ *Some hf* $\wedge$ *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C'* (*s1'*, *h''*) *S*
proof −
fix *H hf C' s1' h'*
assume *asm0*: Δ = *None*
Some H = *Some h* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H* $\wedge$ *red C* (*s1*, *FractionalHeap.normalize* (*get-fh H*)) *C'* (*s1'*, *h'*)
then obtain *s0'* **where** *red C* (*s0*, *FractionalHeap.normalize* (*get-fh H*)) *C'* (*s0'*, *h'*) *agrees* (*fvC C* $\cup$ *vars*) *s1' s0'*
using *red-agrees*[*of C* (*s1*, *FractionalHeap.normalize* (*get-fh H*)) *C'* (*s1'*, *h'*) *fvC C* $\cup$ *vars*]
using *Suc.prems*(*2*) *agrees-def fst-conv snd-conv sup-ge1*
by (*metis* (*mono-tags*, *lifting*))
then obtain *h'' H'* **where**
r: *full-ownership* (*get-fh H'*) $\wedge$ *no-guard H'* $\wedge$ *h'* = *FractionalHeap.normalize* (*get-fh H'*) $\wedge$ *Some H'* = *Some h''* $\oplus$ *Some hf* $\wedge$ *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C'* (*s0'*, *h''*) *S*
using *Suc.prems*(*1*) *asm0*(*1*) *asm0*(*2*) *safeNoneE*(*3*) **by** *blast*
then have *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C'* (*s1'*, *h''*) *S*
using *Suc.hyps*[*of C' s0' h'' s1'*]
using ‹*agrees* (*fvC C* $\cup$ *vars*) *s1' s0'*› *agrees-union asm0*(*1*) *asm0*(*2*)

assms(3) option.distinct(1) red-properties(1)
by (*metis* (*mono-tags*, *lifting*) *agrees-def subset-iff*)
then show $\exists h''\ H'$. *full-ownership* (*get-fh* H') $\wedge$ *no-guard* $H' \wedge h' =$ *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* $H' =$ *Some* $h'' \oplus$ *Some hf* $\wedge$ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (*s1*', h'') *S*
using *r* **by** *blast*
qed
show *accesses C s1* $\subseteq$ *dom* (*fst h*) $\wedge$ *writes C s1* $\subseteq$ *fpdom* (*fst h*)
by (*metis Suc.prems(1) Suc.prems(2) accesses-agrees writes-agrees agrees-union fst-conv safeAccessesE snd-conv*)
fix *H hf* C' *s1*' h' *hj v0* Γ
assume *asm0*: $\Delta =$ *Some* Γ
Some H = *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s1 hj* $\Gamma \wedge$ *red C* (*s1*, *normalize* (*get-fh H*)) C' (*s1*', h')
then obtain *s0*' **where** *red C* (*s0*, *FractionalHeap.normalize* (*get-fh H*)) C' (*s0*', h') *agrees* (*fvC C* $\cup$ *vars* $\cup$ *fvA* (*invariant* Γ)) *s1*' *s0*'
using *red-agrees*[*of C* (*s1*, *FractionalHeap.normalize* (*get-fh H*)) C' (*s1*', h') *fvC C* $\cup$ *vars* $\cup$ *fvA* (*invariant* Γ)]
using *Suc.prems(2) Suc.prems(4) agrees-comm agrees-union fst-conv snd-conv sup-assoc sup-ge1*
by (*metis* (*no-types*, *lifting*))
moreover have *sat-inv s0 hj* Γ
using *Suc.prems(4) agrees-comm asm0(1) asm0(2) sat-inv-agrees* **by** *blast*
ultimately obtain h'' H' *hj*' **where** *r*: *full-ownership* (*get-fh* H') $\wedge$ *semi-consistent* Γ *v0* $H' \wedge$ *sat-inv s0*' *hj*' Γ
$\wedge$ $h' =$ *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* $H' =$ *Some* $h'' \oplus$ *Some hj*' $\oplus$ *Some hf* $\wedge$ *safe n* (*Some* Γ) C' (*s0*', h'') *S*
using *Suc.prems(1) asm0(1) asm0(2) safeSomeE(3)*[*of n* Γ *C s0 h S H hj hf*]
by *blast*
then have *sat-inv s1*' *hj*' Γ
using ‹*agrees* (*fvC C* $\cup$ *vars* $\cup$ *fvA* (*invariant* Γ)) *s1*' *s0*'› *agrees-comm agrees-union sat-inv-agrees* **by** *blast*
moreover have *safe n* (*Some* Γ) C' (*s1*', h'') *S*
using *Suc.hyps*[*of* C' *s0*' h'' *s1*'] ‹*agrees* (*fvC C* $\cup$ *vars* $\cup$ *fvA* (*invariant* Γ)) *s1*' *s0*'› ‹*red C* (*s0*, *FractionalHeap.normalize* (*get-fh H*)) C' (*s0*', h')›
agrees-def agrees-union asm0(1) assms(3) option.inject r red-properties
by (*metis* (*mono-tags*, *lifting*) *subset-Un-eq*)
ultimately show $\exists h''\ H'$ *hj*'.
full-ownership (*get-fh* H') $\wedge$
semi-consistent Γ *v0* $H' \wedge$
sat-inv s1' *hj*' $\Gamma \wedge h' =$ *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* $h'' \oplus$ *Some hj*' $\oplus$ *Some hf* $\wedge$ *safe n* (*Some* Γ) C' (*s1*', h'') *S*
using *r* **by** *blast*
qed
qed (*simp*)

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*
  by (*meson assms*(*1*) *assms*(*2*) *assms*(*3*) *not-Some-eq safe-free-vars-aux*)

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*
  by (*metis agrees-union assms*(*1*) *assms*(*2*) *assms*(*3*) *option.inject safe-free-vars-aux*)

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 $\bigwedge\Gamma.\ \Delta = \mathit{Some}\ \Gamma \Longrightarrow \mathit{agrees}\ (\mathit{fvA}\ (\mathit{invariant}\ \Gamma))\ s\ s'$
  shows *safe n* Δ *C* (*s'*, *h*) *S*
proof (*cases* Δ)
  case *None*
  then show *?thesis*
    using *assms*(*1*) *assms*(*2*) *assms*(*3*) *safe-free-vars-None* **by** *blast*
next
  case (*Some* Γ)
  then show *?thesis*
    using *agrees-union assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *safe-free-vars-Some*
by *blast*
qed

lemma *restrict-safe-to-bounded*:
  assumes *safe n* Δ *C* (*s*, *h*) *S*
    and *bounded h*
  shows *safe n* Δ *C* (*s*, *h*) (*Set.filter* (*bounded* ∘ *snd*) *S*)
  using *assms*
proof (*induct n arbitrary*: *s h C*)
  case (*Suc n*)
  show *?case*
  proof (*rule safeI*)
    have $C = \mathit{Cskip} \Longrightarrow (s, h) \in S$
      using *Suc.prems*(*1*) *safe.elims*(*2*) **by** *blast*
    then show $C = \mathit{Cskip} \Longrightarrow (s, h) \in \mathit{Set.filter}\ (\mathit{bounded} \circ \mathit{snd})\ S$
      by (*simp add*: *Suc.prems*(*2*))
    show *no-abort* Δ *C s h* **using** *Suc.prems*(*1*) *safe.elims*(*2*) **by** *blast*
    show $\mathit{accesses}\ C\ s \subseteq \mathit{dom}\ (\mathit{fst}\ h) \land \mathit{writes}\ C\ s \subseteq \mathit{fpdom}\ (\mathit{fst}\ h)$
      by (*metis Suc.prems*(*1*) *fst-conv safeAccessesE snd-conv*)

fix H hf C' s' h'

assume *asm0*: Δ = *None* *Some* H = *Some* h $\oplus$ *Some* hf $\wedge$ *full-ownership* (*get-fh* H) $\wedge$ *no-guard* H $\wedge$ *red* C (s, *FractionalHeap.normalize* (*get-fh* H)) C' (s', h')

then obtain h'' H' **where** *full-ownership* (*get-fh* H') $\wedge$
no-guard H' $\wedge$
h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some* hf $\wedge$ *safe* n *None* C' (s', h'') S

using *Suc.prems*(*1*) *safeNoneE*(*3*) **by** *blast*

then have *safe* n *None* C' (s', h'') (*Set.filter* (*bounded* $\circ$ *snd*) S)

using *Suc*(*1*)[*of* C' s' h''] **apply** *simp*

using ‹⟦*safe* n Δ C' (s', h'') S; *bounded* h'⟧ $\Longrightarrow$ *safe* n Δ C' (s', h'') (*Set.filter* (*bounded* $\circ$ *snd*) S)› ‹*full-ownership* (*get-fh* H') $\wedge$ *no-guard* H' $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some* hf $\wedge$ *safe* n *None* C' (s', h'') S› *asm0*(*1*) *bounded-smaller-sum* *full-ownership-then-bounded* **by** *blast*

then show $\exists$ h'' H'.
full-ownership (*get-fh* H') $\wedge$
no-guard H' $\wedge$
h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some* hf $\wedge$ *safe* n *None* C' (s', h'') (*Set.filter* (*bounded* $\circ$ *snd*) S)

using ‹*full-ownership* (*get-fh* H') $\wedge$ *no-guard* H' $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some* hf $\wedge$ *safe* n *None* C' (s', h'') S› **by** *blast*

next

fix H hf C' s' h' hj *v0* Γ

assume *asm0*: Δ = *Some* Γ *Some* H = *Some* h $\oplus$ *Some* hj $\oplus$ *Some* hf $\wedge$
full-ownership (*get-fh* H) $\wedge$ *semi-consistent* Γ *v0* H $\wedge$ *sat-inv* s hj Γ $\wedge$ *red* C (s, *FractionalHeap.normalize* (*get-fh* H)) C' (s', h')

then obtain h'' H' hj' **where** *full-ownership* (*get-fh* H') $\wedge$
semi-consistent Γ *v0* H' $\wedge$
sat-inv s' hj' Γ $\wedge$
h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some* hj' $\oplus$ *Some* hf $\wedge$ *safe* n (*Some* Γ) C' (s', h'') S

using *Suc.prems*(*1*) *safeSomeE*(*3*) **by** *blast*

then have *safe* n (*Some* Γ) C' (s', h'') (*Set.filter* (*bounded* $\circ$ *snd*) S)

using *Suc*(*1*)[*of* C' s' h''] **apply** *simp*

by (*metis* (*no-types*, *opaque-lifting*) ‹⟦*safe* n Δ C' (s', h'') S; *bounded* h'⟧ $\Longrightarrow$ *safe* n Δ C' (s', h'') (*Set.filter* (*bounded* $\circ$ *snd*) S)› ‹*full-ownership* (*get-fh* H') $\wedge$ *semi-consistent* Γ *v0* H' $\wedge$ *sat-inv* s' hj' Γ $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some* hj' $\oplus$ *Some* hf $\wedge$ *safe* n (*Some* Γ) C' (s', h'') S› *asm0*(*1*) *bounded-smaller* *full-ownership-then-bounded* *larger3* *plus-comm*)

then show $\exists$ h'' H' hj'.
full-ownership (*get-fh* H') $\wedge$
semi-consistent Γ *v0* H' $\wedge$
sat-inv s' hj' Γ $\wedge$
h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some* hj' $\oplus$ *Some* hf $\wedge$ *safe* n (*Some* Γ) C' (s', h'') (*Set.filter* (*bounded* $\circ$ *snd*) S)

using ‹*full-ownership* (*get-fh* H') ∧ *semi-consistent* Γ *v0* H' ∧ *sat-inv* s' hj' Γ ∧ h' = *FractionalHeap.normalize* (*get-fh* H') ∧ *Some* H' = *Some* h'' ⊕ *Some* hj' ⊕ *Some hf* ∧ *safe n* (*Some* Γ) C' (s', h'') *S*› **by** *blast*

qed
qed (*simp*)

4.3.4 Hoare triples

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

definition *hoare-triple-valid* :: ('*i*, '*a*, *nat*) *cont* ⇒ ('*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* ∧ *bounded* (*snd* σ) ⟶ *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*
by (*metis assms*(*1*) *assms*(*2*) *hoare-triple-valid-def prod.collapse*)

lemma *hoare-triple-validI-bounded*:
assumes ⋀*s h n*. (*s*, *h*), (*s*, *h*) ⊨ *P* ⟹ *bounded h* ⟹ *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*
by (*metis assms*(*1*) *assms*(*2*) *hoare-triple-valid-def prod.collapse*)

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*
using *assms hoare-triple-valid-def* **by** *metis*

lemma *hoare-triple-valid-smallerI-bounded*:
assumes ⋀σ *n*. σ, σ ⊨ *P* ⟹ *bounded* (*snd* σ) ⟹ *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*
using *assms hoare-triple-valid-def* **by** *metis*

lemma *hoare-triple-validE*:
assumes *hoare-triple-valid* Γ *P C Q*
shows ∃Σ. (∀σ *n*. σ, σ ⊨ *P* ∧ *bounded* (*snd* σ) ⟶ *safe n* Γ *C* σ (Σ σ)) ∧
(∀σ σ'. σ, σ' ⊨ *P* ⟶ *pair-sat* (Σ σ) (Σ σ') *Q*)
using *assms hoare-triple-valid-def* **by** *blast*

lemma *hoare-triple-valid-simplerE*:

assumes *hoare-triple-valid* Γ P C Q
and σ, $\sigma' \models P$
and *bounded* (*snd* σ)
and *bounded* (*snd* σ')
shows $\exists S\ S'.$ *safe* $n\ \Gamma\ C\ \sigma\ S \wedge$ *safe* $n\ \Gamma\ C\ \sigma'\ S' \wedge$ *pair-sat* $S\ S'\ Q$
by (*meson always-sat-refl assms hoare-triple-validE sat-comm*)

end

4.4 Soundness of the Rules

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

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

theory *Soundness*
imports *Safety AbstractCommutativity*
begin

4.4.1 Skip

lemma *safe-skip*:
fixes Δ :: (*'i*, *'a*, *nat*) *cont*
assumes (s, h) $\in S$
shows *safe* n Δ *Cskip* (s, h) S
using *assms*
proof (*induct n*)
case (*Suc n*)
then show *?case*
proof (*cases* Δ)
case *None*
then show *?thesis*
by (*simp add*: *Suc.prems*)
next
case (*Some a*)
then show *?thesis*
by (*simp add*: *assms*)
qed
qed (*simp*)

theorem *rule-skip*:
hoare-triple-valid Γ *P Cskip P*
proof (*rule hoare-triple-validI*)
let $?\Sigma = \lambda\sigma.\ \{\sigma\}$
show $\bigwedge s\ h\ n.\ (s,\ h),\ (s,\ h) \models P \Longrightarrow safe\ n\ \Gamma\ Cskip\ (s,\ h)\ (?\Sigma\ (s,\ h))$
by (*simp add*: *safe-skip*)
show $\bigwedge s\ h\ s'\ h'.\ (s,\ h),\ (s',\ h') \models P \Longrightarrow pair\text{-}sat\ \{(s,\ h)\}\ \{(s',\ h')\}\ P$
by (*metis pair-sat-smallerI singleton-iff*)
qed

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 $\bigwedge\Gamma.\ \Delta = Some\ \Gamma \Longrightarrow x \notin fvA\ (invariant\ \Gamma)$
shows *safe m* Δ (*Cassign x E*) (*s*, *h*) { (*s*(*x* := *edenot E s*), *h*) }
proof (*induct m*)
case (*Suc n*)
show *safe* (*Suc n*) Δ (*Cassign x E*) (*s*, *h*) {(*s*(*x* := *edenot E s*), *h*)}
proof (*rule safeI*)
show *no-abort* Δ (*Cassign x E*) *s h*
using *aborts-assign-cases no-abortI* **by** *blast*

show $\bigwedge H\ hf\ C'\ s'\ h'.$
$\Delta = None \Longrightarrow$
Some H = *Some h* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H* $\wedge$ *red* (*Cassign x E*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) C' (s', h') $\Longrightarrow$
$\exists h''\ H'.$
full-ownership (*get-fh* H') $\wedge$
no-guard H' $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some hf* $\wedge$ *safe n None* C' (s', h'') {(*s*(*x* := *edenot E s*), *h*)}
by (*metis Pair-inject insertI1 red-assign-cases safe-skip*)
show *accesses* (*Cassign x E*) *s* $\subseteq$ *dom* (*fst h*) $\wedge$ *writes* (*Cassign x E*) *s* $\subseteq$ *fpdom* (*fst h*)
by *simp*
fix *H hf* C' s' h' *hj v0* Γ
assume *asm0*: Δ = *Some* Γ *Some H* = *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$
full-ownership (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s hj* Γ $\wedge$ *red* (*Cassign x E*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) C' (s', h')
then have *sat-inv* (*s*(*x* := *edenot E s*)) *hj* Γ
by (*meson agrees-update assms sat-inv-agrees*)
then show $\exists h''\ H'\ hj'.$ *full-ownership* (*get-fh* H') $\wedge$ *semi-consistent* Γ *v0* H' $\wedge$ *sat-inv* s' hj' Γ $\wedge$
h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some*

hj′ ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C′* (*s′*, *h′′*) {(*s*(*x* := *edenot E s*), *h*)}
by (*metis* (*no-types*, *lifting*) *asm0*(*2*) *insertI1 old.prod.inject red-assign-cases safe-skip*)
qed (*simp*)
qed (*simp*)

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 −
define Σ :: *store* × (′*i*, ′*a*) *heap* ⇒ (*store* × (′*i*, ′*a*) *heap*) *set* **where** Σ = (λσ. { ((*fst* σ)(*x* := *edenot E* (*fst* σ)), *snd* σ) })

show *?thesis*
proof (*rule hoare-triple-validI*)
show ⋀*s h n*. (*s*, *h*), (*s*, *h*) ⊨ *subA x E P* ⟹ *safe n* Δ (*Cassign x E*) (*s*, *h*) (Σ (*s*, *h*))
using *assms safe-assign* **by** (*metis* Σ*-def fst-eqD snd-eqD*)
show ⋀*s h s′ h′*. (*s*, *h*), (*s′*, *h′*) ⊨ *subA x E P* ⟹ *pair-sat* (Σ (*s*, *h*)) (Σ (*s′*, *h′*)) *P*
by (*metis assms*(*2*) Σ*-def fst-conv pair-sat-smallerI singleton-iff snd-conv subA-assign*)
qed
qed

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 (*induct n*)
case (*Suc n*)
show *?case*
proof (*rule safeNoneI*)
show *Calloc x E* = *Cskip* ⟹ (*s*, *Map.empty*, *gs*, *gu*) ∈ {(*s*(*x* := *a*), [*a* ↦ (*pwrite*, *edenot E s*)], *gs*, *gu*) |*a*. *True*} **by** *simp*
show *no-abort None* (*Calloc x E*) *s* (*Map.empty*, *gs*, *gu*)
using *aborts-alloc-cases no-abort.simps*(*1*) **by** *blast*
fix *H hf C′ s′ h′*
assume *asm0*: *Some H* = *Some* (*Map.empty*, *gs*, *gu*) ⊕ *Some hf* ∧
full-ownership (*get-fh H*) ∧ *no-guard H* ∧ *red* (*Calloc x E*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)

show $\exists h''\ H'$.
full-ownership (*get-fh* H') $\wedge$
no-guard H' $\wedge$
h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$
Some H' = *Some* h'' $\oplus$ *Some hf* $\wedge$ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (s', h'') {(s(x := a), [$a \mapsto$ (*pwrite*, *edenot E s*)], *gs*, *gu*) |*a*. *True*}
proof (*rule red-alloc-cases*)
show *red* (*Calloc x E*) (s, *FractionalHeap.normalize* (*get-fh H*)) C' (s', h')
using *asm0* **by** *blast*
fix *sa h v*
assume *asm1*: (s, *FractionalHeap.normalize* (*get-fh H*)) = (*sa*, *h*) C' = *Cskip* (s', h') = (*sa*(x := v), *h*($v \mapsto$ *edenot E sa*))
$v \notin$ *dom h*
then have $v \notin$ *dom* (*get-fh H*)
by (*simp add*: *dom-normalize*)
then have $v \notin$ *dom* (*get-fh hf*)
by (*metis asm0 fst-conv get-fh.simps no-guard-and-no-heap no-guard-then-smaller-same no-guards-remove plus-comm*)

moreover have (*Map.empty*($v \mapsto$ (*pwrite*, *edenot E sa*)), *gs*, *gu*) ## *hf*
proof (*rule compatibleI*)
show *compatible-fract-heaps* (*get-fh* ([$v \mapsto$ (*pwrite*, *edenot E sa*)], *gs*, *gu*)) (*get-fh hf*)
proof (*rule compatible-fract-heapsI*)
fix *l p* p'
assume *asm0*: *get-fh* ([$v \mapsto$ (*pwrite*, *edenot E sa*)], *gs*, *gu*) *l* = *Some p* $\wedge$ *get-fh hf l* = *Some* p'
then show *pgte pwrite* (*padd* (*fst p*) (*fst* p'))
by (*metis calculation domIff fst-conv fun-upd-other get-fh.elims option.distinct*(*1*))
show *snd p* = *snd* p'
by (*metis asm0 calculation domIff fst-conv fun-upd-other get-fh.elims option.distinct*(*1*))
qed
show $\bigwedge k$. *get-gu* ([$v \mapsto$ (*pwrite*, *edenot E sa*)], *gs*, *gu*) *k* = *None* $\vee$ *get-gu hf k* = *None*
by (*metis asm0 compatible-def compatible-eq get-gu.simps option.discI snd-conv*)
show $\bigwedge p\ p'$. *get-gs* ([$v \mapsto$ (*pwrite*, *edenot E sa*)], *gs*, *gu*) = *Some p* $\wedge$ *get-gs hf* = *Some* p' $\Longrightarrow$ *pgte pwrite* (*padd* (*fst p*) (*fst* p'))
by (*metis asm0 no-guard-def no-guard-then-smaller-same option.simps*(*3*) *plus-comm*)
qed
then obtain H' **where** *Some* H' = *Some* (*Map.empty*($v \mapsto$ (*pwrite*, *edenot E sa*)), *gs*, *gu*) $\oplus$ *Some hf*
by *auto*
moreover have (s', (*Map.empty*($v \mapsto$ (*pwrite*, *edenot E sa*)), *gs*, *gu*)) $\in$ {(s(x := a), [$a \mapsto$ (*pwrite*, *edenot E s*)], *gs*, *gu*) |*a*. *True*}

using *asm1*(*1*) *asm1*(*3*) **by** *blast*
then have *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C'* (*s'*, (*Map.empty*(*v* ↦ (*pwrite*, *edenot E sa*)), *gs*, *gu*)) {(*s*(*x* := *a*), [*a* ↦ (*pwrite*, *edenot E s*)], *gs*, *gu*) |*a*. *True*}
by (*simp add*: *asm1*(*2*) *safe-skip*)
moreover have *full-ownership* (*get-fh H'*) ∧ *no-guard H'* ∧ *h'* = *FractionalHeap.normalize* (*get-fh H'*)
proof −
have *full-ownership* (*get-fh H'*)
proof (*rule full-ownershipI*)
fix *l p*
assume *get-fh H' l* = *Some p*
show *fst p* = *pwrite*
proof (*cases l* = *v*)
case *True*
then have *get-fh hf l* = *None*
using *calculation*(*1*) **by** *blast*
then have *get-fh H' l* = (*Map.empty*(*v* ↦ (*pwrite*, *edenot E sa*))) *l*
by (*metis calculation*(*2*) *fst-conv get-fh.simps sum-second-none-get-fh*)
then show *?thesis*
using *True* ‹*get-fh H' l* = *Some p*› **by** *fastforce*
next
case *False*
then have *get-fh* ([*v* ↦ (*pwrite*, *edenot E sa*)], *gs*, *gu*) *l* = *None*
by *simp*
then show *fst p* = *pwrite*
by (*metis* (*mono-tags*, *lifting*) ‹*get-fh H' l* = *Some p*› *asm0 calculation*(*2*) *fst-conv full-ownership-def get-fh.elims sum-first-none-get-fh*)
qed
qed
moreover have *no-guard H'*
proof −
have *no-guard hf*
by (*metis asm0 no-guard-then-smaller-same plus-comm*)
moreover have *no-guard* (*Map.empty*, *gs*, *gu*)
using *asm0 no-guard-then-smaller-same* **by** *blast*
ultimately show *?thesis*
by (*metis* ‹*Some H'* = *Some* ([*v* ↦ (*pwrite*, *edenot E sa*)], *gs*, *gu*) ⊕ *Some hf*› *decompose-heap-triple no-guard-remove*(*1*) *no-guard-remove*(*2*) *no-guards-remove remove-guards-def snd-conv*)
qed
moreover have *h'* = *FractionalHeap.normalize* (*get-fh H'*)
proof (*rule ext*)
fix *l* **show** *h' l* = *FractionalHeap.normalize* (*get-fh H'*) *l*
proof (*cases l* = *v*)
case *True*
then have *get-fh* (*Map.empty*(*v* ↦ (*pwrite*, *edenot E sa*)), *gs*, *gu*) *l* = *Some* (*pwrite*, *edenot E sa*)
by *auto*
then have *get-fh hf l* = *None*

using *True* ‹$v \notin$ *dom* (*get-fh hf*)› **by** *force*
then show h' l = *FractionalHeap.normalize* (*get-fh* H') l
apply (*cases* h' l)
using *True asm1*(*3*) **apply** *auto*[*1*]
by (*metis* (*no-types, lifting*) *FractionalHeap.normalize-def True* ‹*Some* H' = *Some* ([$v \mapsto$ (*pwrite, edenot E sa*)], *gs, gu*) $\oplus$ *Some hf*› ‹*get-fh* ([$v \mapsto$ (*pwrite, edenot E sa*)], *gs, gu*) l = *Some* (*pwrite, edenot E sa*)› *apply-opt.simps*(*2*) *asm1*(*3*) *fun-upd-same snd-conv sum-second-none-get-fh*)
next
case *False*
then have *get-fh* (*Map.empty*($v \mapsto$ (*pwrite, edenot E sa*)), *gs, gu*) l = *None*
by *simp*
then have *get-fh* H' l = *get-fh hf* l
using ‹*Some* H' = *Some* ([$v \mapsto$ (*pwrite, edenot E sa*)], *gs, gu*) $\oplus$ *Some hf*› *sum-first-none-get-fh* **by** *blast*
moreover have *get-fh H l* = *get-fh hf l*
by (*metis asm0 fst-conv get-fh.elims plus-comm sum-second-none-get-fh*)
ultimately show *?thesis*
proof (*cases get-fh hf l*)
case *None*
then show *?thesis*
by (*metis False FractionalHeap.normalize-eq*(*1*) ‹*get-fh H l* = *get-fh hf l*› ‹*get-fh* H' l = *get-fh hf l*› *asm1*(*1*) *asm1*(*3*) *fun-upd-apply old.prod.inject*)
next
case (*Some f*)
then show *?thesis*
by (*metis* (*no-types, lifting*) *False FractionalHeap.normalize-eq*(*1*) *FractionalHeap.normalize-eq*(*2*) ‹*get-fh H l* = *get-fh hf l*› ‹*get-fh* H' l = *get-fh hf l*› *asm1*(*1*) *asm1*(*3*) *domD not-in-dom fun-upd-apply old.prod.inject*)
qed
qed
qed
ultimately show *?thesis*
by *auto*
qed
ultimately show $\exists h''$ H'. *full-ownership* (*get-fh* H') $\wedge$ *no-guard* H' $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some hf* $\wedge$ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (s', h'') {(*s*(*x* := *a*), [*a* $\mapsto$ (*pwrite, edenot E s*)], *gs, gu*) |*a. True*}
by *blast*
qed
qed (*simp*)
qed (*simp*)

lemma *safe-new-Some*:
assumes $x \notin$ *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 (*induct n*)
case (*Suc n*)
show *?case*
proof (*rule safeSomeI*)
show *Calloc x E* = *Cskip* ⟹ (*s*, *Map.empty*, *gs*, *gu*) ∈ {(*s*(*x* := *a*), [*a* ↦ (*pwrite*, *edenot E s*)], *gs*, *gu*) |*a*. *True*} **by** *simp*
show *no-abort* (*Some* Γ) (*Calloc x E*) *s* (*Map.empty*, *gs*, *gu*)
using *aborts-alloc-cases no-abort.simps*(*2*) **by** *blast*
fix *H hf C′ s′ h′ hj v0*
assume *asm0*: *Some H* = *Some* (*Map.empty*, *gs*, *gu*) ⊕ *Some hj* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *semi-consistent* Γ *v0 H* ∧ *sat-inv s hj* Γ ∧ *red* (*Calloc x E*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)

then obtain *hjf* **where** *Some hjf* = *Some hj* ⊕ *Some hf*
by (*metis plus.simps*(*2*) *plus.simps*(*3*) *plus-asso*)
then have *Some H* = *Some* (*Map.empty*, *gs*, *gu*) ⊕ *Some hjf*
by (*metis asm0 plus-asso*)

show ∃ *h″ H′ hj′*.
full-ownership (*get-fh H′*) ∧
semi-consistent Γ *v0 H′* ∧
sat-inv s′ hj′ Γ ∧
h′ = *FractionalHeap.normalize* (*get-fh H′*) ∧
Some H′ = *Some h″* ⊕ *Some hj′* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C′* (*s′*, *h″*) {(*s*(*x* := *a*), [*a* ↦ (*pwrite*, *edenot E s*)], *gs*, *gu*) |*a*. *True*}
proof (*rule red-alloc-cases*)
show *red* (*Calloc x E*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)
using *asm0* **by** *blast*
fix *sa h v*
assume *asm1*: (*s*, *FractionalHeap.normalize* (*get-fh H*)) = (*sa*, *h*) *C′* = *Cskip* (*s′*, *h′*) = (*sa*(*x* := *v*), *h*(*v* ↦ *edenot E sa*))
v ∉ *dom h*
then have *v* ∉ *dom* (*get-fh H*)
by (*simp add*: *dom-normalize*)
then have *v* ∉ *dom* (*get-fh hjf*)
by (*metis* (*no-types*, *lifting*) ‹*Some H* = *Some* (*Map.empty*, *gs*, *gu*) ⊕ *Some hjf*› *addition-smaller-domain in-mono plus-comm*)

moreover have (*Map.empty*(*v* ↦ (*pwrite*, *edenot E sa*)), *gs*, *gu*) ## *hjf*
proof (*rule compatibleI*)
show *compatible-fract-heaps* (*get-fh* ([*v* ↦ (*pwrite*, *edenot E sa*)], *gs*, *gu*)) (*get-fh hjf*)
proof (*rule compatible-fract-heapsI*)
fix *l p p′*
assume *asm2*: *get-fh* ([*v* ↦ (*pwrite*, *edenot E sa*)], *gs*, *gu*) *l* = *Some p* ∧ *get-fh hjf l* = *Some p′*
then show *pgte pwrite* (*padd* (*fst p*) (*fst p′*))

by (*metis calculation domIff fst-conv fun-upd-other get-fh.elims option.distinct*(*1*))
show *snd p* = *snd p'*
using *asm2 calculation domIff fst-conv fun-upd-other get-fh.elims option.distinct*(*1*) **by** *metis*
qed
show $\bigwedge k$. *get-gu* ([*v* ↦ (*pwrite*, *edenot E sa*)], *gs*, *gu*) *k* = *None* ∨ *get-gu hjf k* = *None*
by (*metis* ‹*Some H* = *Some* (*Map.empty*, *gs*, *gu*) ⊕ *Some hjf*› *compatible-def compatible-eq get-gu.simps option.discI snd-conv*)
show $\bigwedge p\ p'$. *get-gs* ([*v* ↦ (*pwrite*, *edenot E sa*)], *gs*, *gu*) = *Some p* ∧ *get-gs hjf* = *Some p'* ⟹ *pgte pwrite* (*padd* (*fst p*) (*fst p'*))
by (*metis* ‹*Some H* = *Some* (*Map.empty*, *gs*, *gu*) ⊕ *Some hjf*› *compatible-def compatible-eq get-gs.simps option.simps*(*3*) *snd-eqD*)
qed
then obtain *H'* **where** *Some H'* = *Some* (*Map.empty*(*v* ↦ (*pwrite*, *edenot E sa*)), *gs*, *gu*) ⊕ *Some hjf*
by *auto*
moreover have (*s'*, (*Map.empty*(*v* ↦ (*pwrite*, *edenot E sa*)), *gs*, *gu*)) ∈ {(*s*(*x* := *a*), [*a* ↦ (*pwrite*, *edenot E s*)], *gs*, *gu*) |*a*. *True*}
using *asm1*(*1*) *asm1*(*3*) **by** *blast*
then have *safe n* (*Some* Γ) *C'* (*s'*, (*Map.empty*(*v* ↦ (*pwrite*, *edenot E sa*)), *gs*, *gu*)) {(*s*(*x* := *a*), [*a* ↦ (*pwrite*, *edenot E s*)], *gs*, *gu*) |*a*. *True*}
by (*simp add*: *asm1*(*2*) *safe-skip*)

moreover have *full-ownership* (*get-fh H'*) ∧ *semi-consistent* Γ *v0 H'* ∧ *h'* = *FractionalHeap.normalize* (*get-fh H'*)
proof −
have *full-ownership* (*get-fh H'*)
proof (*rule full-ownershipI*)
fix *l p*
assume *get-fh H' l* = *Some p*
show *fst p* = *pwrite*
proof (*cases l* = *v*)
case *True*
then have *get-fh hjf l* = *None*
using *calculation*(*1*) **by** *blast*
then have *get-fh H' l* = (*Map.empty*(*v* ↦ (*pwrite*, *edenot E sa*))) *l*
by (*metis calculation*(*2*) *fst-conv get-fh.simps sum-second-none-get-fh*)
then show *?thesis*
using *True* ‹*get-fh H' l* = *Some p*› **by** *fastforce*
next
case *False*
then have *get-fh H' l* = *get-fh hjf l* **using** *sum-first-none-get-fh*[*of H' - hjf l*]
using *calculation*(*2*) **by** *force*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) ‹*Some H* = *Some* (*Map.empty*, *gs*, *gu*) ⊕ *Some hjf*› ‹*get-fh H' l* = *Some p*› *asm0 fst-conv full-ownership-def get-fh.elims*

plus-comm sum-second-none-get-fh)
qed
qed
moreover have *h′* = *FractionalHeap.normalize* (*get-fh H′*)
proof (*rule ext*)
fix *l* **show** *h′ l* = *FractionalHeap.normalize* (*get-fh H′*) *l*
proof (*cases l* = *v*)
case *True*
then have *get-fh* (*Map.empty*(*v* ↦ (*pwrite*, *edenot E sa*)), *gs*, *gu*) *l* = *Some* (*pwrite*, *edenot E sa*)
by *auto*
then have *get-fh hjf l* = *None*
using *True* ‹*v* ∉ *dom* (*get-fh hjf*)› **by** *force*
then show *?thesis*
apply (*cases h′ l*)
using *True asm1*(*3*) **apply** *auto*[*1*]
by (*metis* (*no-types*, *lifting*) *FractionalHeap.normalize-def True* ‹*Some H′* = *Some* ([*v* ↦ (*pwrite*, *edenot E sa*)], *gs*, *gu*) ⊕ *Some hjf*› ‹*get-fh* ([*v* ↦ (*pwrite*, *edenot E sa*)], *gs*, *gu*) *l* = *Some* (*pwrite*, *edenot E sa*)› *apply-opt.simps*(*2*) *asm1*(*3*) *fun-upd-same snd-conv sum-second-none-get-fh*)
next
case *False*
then have *get-fh* (*Map.empty*(*v* ↦ (*pwrite*, *edenot E sa*)), *gs*, *gu*) *l* = *None*
by *simp*
then have *get-fh H′ l* = *get-fh hjf l*
using ‹*Some H′* = *Some* ([*v* ↦ (*pwrite*, *edenot E sa*)], *gs*, *gu*) ⊕ *Some hjf*› *sum-first-none-get-fh* **by** *blast*
moreover have *get-fh H l* = *get-fh hjf l*
by (*metis* ‹*Some H* = *Some* (*Map.empty*, *gs*, *gu*) ⊕ *Some hjf*› *fst-eqD get-fh.simps sum-first-none-get-fh*)
ultimately show *?thesis*
proof (*cases get-fh hjf l*)
case *None*
then show *?thesis*
by (*metis False FractionalHeap.normalize-eq*(*1*) ‹*get-fh H l* = *get-fh hjf l*› ‹*get-fh H′ l* = *get-fh hjf l*› *asm1*(*1*) *asm1*(*3*) *fun-upd-apply old.prod.inject*)
next
case (*Some f*)
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *False FractionalHeap.normalize-eq*(*1*) *FractionalHeap.normalize-eq*(*2*) ‹*get-fh H l* = *get-fh hjf l*› ‹*get-fh H′ l* = *get-fh hjf l*› *asm1*(*1*) *asm1*(*3*) *domD not-in-dom fun-upd-apply old.prod.inject*)
qed
qed
qed
moreover have *semi-consistent* Γ *v0 H′*
proof (*rule semi-consistentI*)
have *get-gs H′* = *get-gs H*

by (*metis* ‹*Some H* = *Some* (*Map.empty*, *gs*, *gu*) ⊕ *Some hjf*› ‹*Some H'* = *Some* ([*v* ↦ (*pwrite*, *edenot E sa*)], *gs*, *gu*) ⊕ *Some hjf*› *fst-conv get-gs.simps option.discI option.sel plus.simps*(*3*) *snd-conv*)
moreover have *get-gu H'* = *get-gu H*
by (*metis* ‹*Some H* = *Some* (*Map.empty*, *gs*, *gu*) ⊕ *Some hjf*› ‹*Some H'* = *Some* ([*v* ↦ (*pwrite*, *edenot E sa*)], *gs*, *gu*) ⊕ *Some hjf*› *get-gu.simps option.discI option.sel plus.simps*(*3*) *snd-conv*)
ultimately show *all-guards H'*
by (*metis all-guards-def asm0 semi-consistent-def*)
show *reachable* Γ *v0 H'*
proof (*rule reachableI*)
fix *sargs uargs*
assume *get-gs H'* = *Some* (*pwrite*, *sargs*) ∧ (∀ *k*. *get-gu H' k* = *Some* (*uargs k*))
then have *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs uargs* (*view* Γ (*FractionalHeap.normalize* (*get-fh H*)))
by (*metis* ‹*get-gs H'* = *get-gs H*› ‹*get-gu H'* = *get-gu H*› *asm0 reachableE semi-consistent-def*)
moreover have *view* Γ (*FractionalHeap.normalize* (*get-fh H*)) = *view* Γ (*FractionalHeap.normalize* (*get-fh H'*))
proof −
have *view* Γ (*FractionalHeap.normalize* (*get-fh H*)) = *view* Γ (*FractionalHeap.normalize* (*get-fh hj*))
using *view-function-of-invE*[*of* Γ *s hj H*] **by** (*simp add*: *asm0 assms*(*2*) *larger3*)
moreover have *view* Γ (*FractionalHeap.normalize* (*get-fh H'*)) = *view* Γ (*FractionalHeap.normalize* (*get-fh hj*))
using *view-function-of-invE*[*of* Γ *s hj H'*]
by (*metis* ‹*Some H'* = *Some* ([*v* ↦ (*pwrite*, *edenot E sa*)], *gs*, *gu*) ⊕ *Some hjf*› ‹*Some hjf* = *Some hj* ⊕ *Some hf*› *asm0 assms*(*2*) *larger3 plus-comm*)
ultimately show *?thesis* **by** *simp*
qed
ultimately show *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs uargs* (*view* Γ (*FractionalHeap.normalize* (*get-fh H'*)))
by *simp*
qed
qed
ultimately show *?thesis*
by *auto*
qed

moreover have *sat-inv s' hj* Γ
proof (*rule sat-invI*)
show *no-guard hj*
using *asm0 sat-inv-def* **by** *blast*
have *agrees* (*fvA* (*invariant* Γ)) *s s'*
using *asm1*(*1*) *asm1*(*3*) *assms*
by (*simp add*: *agrees-update*)
then show (*s'*, *hj*), (*s'*, *hj*) ⊨ *invariant* Γ

using *asm0 sat-inv-agrees sat-inv-def* **by** *blast*
qed

ultimately show $\exists h'' H' hj'$. *full-ownership* (*get-fh* H') $\wedge$ *semi-consistent* Γ *v0* H' $\wedge$ *sat-inv* s' hj' Γ $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$
Some H' = *Some* h'' $\oplus$ *Some* hj' $\oplus$ *Some hf* $\wedge$ *safe n* (*Some* Γ) C' (s', h'') $\{(s(x := a), [a \mapsto (pwrite, edenot\ E\ s)], gs, gu) \mid a.\ True\}$
by (*metis* (*no-types*, *lifting*) ‹*Some hjf* = *Some hj* $\oplus$ *Some hf*› *plus-asso*)
qed
qed (*simp*)
qed (*simp*)

lemma *safe-new*:
fixes Δ :: ($'i$, $'a$, *nat*) *cont*
assumes $\bigwedge\Gamma$. Δ = *Some* $\Gamma \Longrightarrow x \notin$ *fvA* (*invariant* Γ) $\wedge$ *view-function-of-inv* Γ
shows *safe n* Δ (*Calloc x E*) (*s*, (*Map.empty*, *gs*, *gu*)) $\{ (s(x := a), (Map.empty(a \mapsto (pwrite, edenot\ E\ s)), gs, gu)) \mid a.\ True \}$
apply (*cases* Δ)
using *safe-new-None safe-new-Some assms* **by** *blast+*

theorem *new-rule*:
fixes Δ :: ($'i$, $'a$, *nat*) *cont*
assumes $x \notin$ *fvE E*
and $\bigwedge\Gamma$. Δ = *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 (*rule hoare-triple-validI*)
define Σ :: *store* $\times$ ($'i$, $'a$) *heap* $\Rightarrow$ (*store* $\times$ ($'i$, $'a$) *heap*) *set* **where** $\Sigma = (\lambda(s, h).\ \{ (s(x := a), (Map.empty(a \mapsto (pwrite, edenot\ E\ s)), get\text{-}gs\ h, get\text{-}gu\ h)) \mid a.\ True \})$

show $\bigwedge s\ h\ n.\ (s, h), (s, h) \models Emp \Longrightarrow$ *safe n* Δ (*Calloc x E*) (*s*, *h*) (Σ (*s*, *h*))
proof −
fix *s h n* **assume** (*s*, *h* :: ($'i$, $'a$) *heap*), (*s*, *h*) $\models$ *Emp* **then have** *get-fh h* = *Map.empty*
by *simp*
then have *h* = (*Map.empty*, *get-gs h*, *get-gu h*) **using** *decompose-heap-triple*
by *metis*
moreover have *safe n* Δ (*Calloc x E*) (*s*, *Map.empty*, *get-gs h*, *get-gu h*) $\{(s(x := a), [a \mapsto (pwrite, edenot\ E\ s)], get\text{-}gs\ h, get\text{-}gu\ h) \mid a.\ True\}$
using *safe-new assms*(*2*) **by** *blast*
moreover have Σ (*s*, *h*) = $\{ (s(x := a), (Map.empty(a \mapsto (pwrite, edenot\ E\ s)), get\text{-}gs\ h, get\text{-}gu\ h)) \mid a.\ True \}$
using Σ-*def* **by** *force*
ultimately show *safe n* Δ (*Calloc x E*) (*s*, *h*) (Σ (*s*, *h*))
by *presburger*
qed
fix *s1 h1 s2 h2*

assume $(s1, h1 :: ('i, 'a)\ heap), (s2, h2) \models Emp$
show *pair-sat* (*case* (*s1*, *h1*) *of* (*s*, *h*) ⇒ {(*s*(*x* := *a*), [*a* ↦ (*pwrite*, *edenot E s*)], *get-gs h*, *get-gu h*) |*a*. *True*})
(*case* (*s2*, *h2*) *of* (*s*, *h*) ⇒ {(*s*(*x* := *a*), [*a* ↦ (*pwrite*, *edenot E s*)], *get-gs h*, *get-gu h*) |*a*. *True*}) (*PointsTo* (*Evar x*) *pwrite E*)
proof (*rule pair-satI*)
fix *s1′ h1′ s2′ h2′*
assume *asm0*: (*s1′*, *h1′*) ∈ (*case* (*s1*, *h1*) *of* (*s*, *h*) ⇒ {(*s*(*x* := *a*), [*a* ↦ (*pwrite*, *edenot E s*)], *get-gs h*, *get-gu h*) |*a*. *True*}) ∧
(*s2′*, *h2′*) ∈ (*case* (*s2*, *h2*) *of* (*s*, *h*) ⇒ {(*s*(*x* := *a*), [*a* ↦ (*pwrite*, *edenot E s*)], *get-gs h*, *get-gu h*) |*a*. *True*})
then obtain *a1 a2* **where** *s1′* = *s1*(*x* := *a1*) *s2′* = *s2*(*x* := *a2*) *h1′* = ([*a1* ↦ (*pwrite*, *edenot E s1*)], *get-gs h1*, *get-gu h1*)
h2′ = ([*a2* ↦ (*pwrite*, *edenot E s2*)], *get-gs h2*, *get-gu h2*)
by *blast*
then show (*s1′*, *h1′*), (*s2′*, *h2′*) ⊨ *PointsTo* (*Evar x*) *pwrite E*
by (*simp add*: *assms*(*1*))
qed
qed

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*)) }
using *assms*
proof (*induct n*)
case (*Suc n*)
show *?case*
proof (*rule safeNoneI*)
show *Cwrite loc E* = *Cskip* ⟹ (*s*, *fh*, *gs*, *gu*) ∈ {(*s*, *fh*(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *gs*, *gu*)}
by *simp*
show *no-abort None* (*Cwrite loc E*) *s* (*fh*, *gs*, *gu*)
proof (*rule no-abortNoneI*)
fix *hf H* **assume** *asm0*: *Some H* = *Some* (*fh*, *gs*, *gu*) ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *no-guard H*
then have *edenot loc s* ∈ *dom* (*normalize* (*get-fh H*))
by (*metis* (*mono-tags*, *lifting*) *Suc.prems addition-smaller-domain dom-def dom-normalize fst-conv get-fh.simps mem-Collect-eq option.discI subsetD*)
then show ¬ *aborts* (*Cwrite loc E*) (*s*, *normalize* (*get-fh H*))
by (*metis aborts-write-cases fst-eqD snd-eqD*)
qed
show *accesses* (*Cwrite loc E*) *s* ⊆ *dom* (*fst* (*fh*, *gs*, *gu*)) ∧ *writes* (*Cwrite loc E*) *s* ⊆ *fpdom* (*fst* (*fh*, *gs*, *gu*))

using *assms fpdom-def* **by** *fastforce*

fix *H hf C′ s′ h′*
assume *asm0*: *Some H* = *Some* (*fh*, *gs*, *gu*) ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *no-guard H*
∧ *red* (*Cwrite loc E*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)
then have *get-fh hf* (*edenot loc s*) = *None*
proof −
have *compatible-fract-heaps fh* (*get-fh hf*)
by (*metis asm0 compatible-def compatible-eq fst-conv get-fh.elims option.discI*)
then show *?thesis* **using** *compatible-then-dom-disjoint*(*2*)[*of fh get-fh hf*]
assms disjoint-iff-not-equal[*of dom* (*get-fh hf*) *fpdom fh*] *not-in-dom fpdom-def mem-Collect-eq*
by *fastforce*
qed

show ∃ *h″ H′*. *full-ownership* (*get-fh H′*) ∧ *no-guard H′* ∧ *h′* = *FractionalHeap.normalize* (*get-fh H′*)
∧ *Some H′* = *Some h″* ⊕ *Some hf* ∧ *safe n None C′* (*s′*, *h″*) {(*s*, *fh*(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *gs*, *gu*)}
proof (*rule red-write-cases*)
show *red* (*Cwrite loc E*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)
using *asm0* **by** *blast*
fix *sa h*
assume *asm1*: (*s*, *FractionalHeap.normalize* (*get-fh H*)) = (*sa*, *h*) *C′* = *Cskip*
(*s′*, *h′*) = (*sa*, *h*(*edenot loc sa* ↦ *edenot E sa*))
then obtain *s* = *sa h′* = *h*(*edenot loc s* ↦ *edenot E s*) **by** *blast*

let *?h* = (*fh*(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *gs*, *gu*)
have *?h ## hf*
proof (*rule compatibleI*)
show *compatible-fract-heaps* (*get-fh* (*fh*(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *gs*, *gu*)) (*get-fh hf*)
proof (*rule compatible-fract-heapsI*)
fix *l p p′* **assume** *asm2*: *get-fh* (*fh*(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *gs*, *gu*) *l* = *Some p* ∧ *get-fh hf l* = *Some p′*
then show *pgte pwrite* (*padd* (*fst p*) (*fst p′*))
apply (*cases l* = *edenot loc s*)
apply (*metis Suc.prems asm0 fst-conv fun-upd-same get-fh.elims option.sel plus-extract-point-fh*)
by (*metis asm0 fst-conv fun-upd-other get-fh.elims plus-extract-point-fh*)
show *snd p* = *snd p′*
apply (*cases l* = *edenot loc s*)
using ‹*pgte pwrite* (*padd* (*fst p*) (*fst p′*))› *asm2 not-pgte-charact sum-larger*
apply *fastforce*
by (*metis* (*mono-tags*, *opaque-lifting*) *asm0 asm2 fst-eqD get-fh.simps map-upd-Some-unfold plus-extract-point-fh*)
qed

show $\bigwedge k$. *get-gu* (*fh*(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *gs*, *gu*) *k* = *None* ∨ *get-gu hf k* = *None*
by (*metis asm0 compatible-def compatible-eq get-gu.simps option.discI snd-conv*)
show $\bigwedge p\ p'$. *get-gs* (*fh*(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *gs*, *gu*) = *Some p* ∧ *get-gs hf* = *Some p′* ⟹ *pgte pwrite* (*padd* (*fst p*) (*fst p′*))
by (*metis asm0 no-guard-def no-guard-then-smaller-same option.simps*(*3*) *plus-comm*)
qed
then obtain *H′* **where** *Some H′* = *Some ?h* ⊕ *Some hf* **by** *auto*
moreover have *H′* = ((*get-fh H*)(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *get-gs H*, *get-gu H*)
proof (*rule heap-ext*)
show *get-fh H′* = *get-fh* ((*get-fh H*)(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *get-gs H*, *get-gu H*)
using *calculation asm0* **by** (*metis* ‹*get-fh hf* (*edenot loc s*) = *None*› *add-fh-update add-get-fh fst-conv get-fh.simps*)
show *get-gs H′* = *get-gs* ((*get-fh H*)(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *get-gs H*, *get-gu H*)
using *calculation asm0*
by (*metis fst-conv get-gs.simps plus-extract*(*2*) *snd-conv*)
show *get-gu H′* = *get-gu* ((*get-fh H*)(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *get-gs H*, *get-gu H*)
using *add-fh-update*[*of get-fh hf edenot E s fh* (*pwrite*, *edenot E s*)] *asm0 calculation*
by (*metis get-gu.elims plus-extract*(*3*) *snd-conv*)
qed
moreover have *safe n* (*None* :: (*′i*, *′a*, *nat*) *cont*) *C′* (*s′*, *?h*) {(*s*, *fh*(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *gs*, *gu*)}
using ‹*s* = *sa*› *asm1*(*2*) *asm1*(*3*) *safe-skip* **by** *fastforce*
moreover have *full-ownership* (*get-fh H′*) ∧ *no-guard H′* ∧ *h′* = *Fractional-Heap.normalize* (*get-fh H′*)
proof −
have *full-ownership* (*get-fh H′*)
proof (*rule full-ownershipI*)
fix *l p*
assume *asm*: *get-fh H′ l* = *Some p*
then show *fst p* = *pwrite*
proof (*cases l* = *edenot loc s*)
case *True*
then show *?thesis*
using *asm calculation*(*2*) **by** *fastforce*
next
case *False*
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *asm asm0 calculation*(*2*) *fst-eqD full-ownership-def get-fh.simps map-upd-Some-unfold*)
qed
qed

moreover have *no-guard* H' **using** *asm0*
by (*simp add*: ‹$H' = ((get\text{-}fh\ H)(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ get\text{-}gs\ H,\ get\text{-}gu\ H)$› *no-guard-def*)

moreover have $h' = FractionalHeap.normalize\ (get\text{-}fh\ H')$
proof (*rule ext*)
fix l **show** $h'\ l = FractionalHeap.normalize\ (get\text{-}fh\ H')\ l$
proof(*cases* $l = edenot\ loc\ s$)
case *True*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *FractionalHeap.normalize-eq*(*2*) ‹$H' = ((get\text{-}fh\ H)(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ get\text{-}gs\ H,\ get\text{-}gu\ H)$› ‹$h' = h(edenot\ loc\ s \mapsto edenot\ E\ s)$› *fst-conv fun-upd-same get-fh.elims*)
next
case *False*
then have $FractionalHeap.normalize\ (get\text{-}fh\ H')\ l = FractionalHeap.normalize\ (get\text{-}fh\ H)\ l$
using *FractionalHeap.normalize-eq*(*2*)[*of get-fh* H' *l*] *FractionalHeap.normalize-eq*(*2*)[*of get-fh H l*] ‹$H' = ((get\text{-}fh\ H)(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ get\text{-}gs\ H,\ get\text{-}gu\ H)$›
fst-conv fun-upd-other[*of l edenot loc s get-fh H*] *get-fh.simps option.exhaust*
by *metis*
then show *?thesis*
using *False* ‹$h' = h(edenot\ loc\ s \mapsto edenot\ E\ s)$› *asm1*(*1*) **by** *force*
qed
qed
ultimately show *?thesis*
by *auto*
qed
ultimately show $\exists h''\ H'.\ full\text{-}ownership\ (get\text{-}fh\ H') \land no\text{-}guard\ H' \land h' = FractionalHeap.normalize\ (get\text{-}fh\ H') \land$
$Some\ H' = Some\ h'' \oplus Some\ hf \land safe\ n\ None\ C'\ (s',\ h'')\ \{(s,\ fh(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ gs,\ gu)\}$
by (*metis* ‹$Some\ H' = Some\ (fh(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ gs,\ gu) \oplus Some\ hf$› ‹$s = sa$› *asm1*(*2*) *asm1*(*3*) *fst-conv insertI1 safe-skip*)
qed
qed
qed (*simp*)

lemma *safe-write-Some*:
assumes $fh\ (edenot\ loc\ s) = Some\ (pwrite,\ v)$
and *view-function-of-inv* Γ
shows $safe\ n\ (Some\ \Gamma)\ (Cwrite\ loc\ E)\ (s,\ (fh,\ gs,\ gu))\ \{\ (s,\ (fh(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ gs,\ gu))\ \}$
using *assms*
proof (*induct n*)

case (*Suc n*)
show *?case*
proof (*rule safeSomeI*)
show *Cwrite loc E* = *Cskip* $\Longrightarrow$ (*s*, *fh*, *gs*, *gu*) $\in$ {(*s*, *fh*(*edenot loc s* $\mapsto$ (*pwrite*, *edenot E s*)), *gs*, *gu*)}
by *simp*
show *no-abort* (*Some* Γ) (*Cwrite loc E*) *s* (*fh*, *gs*, *gu*)
proof (*rule no-abortSomeI*)
fix *H hf hj v0*
assume *asm0*: *Some H* = *Some* (*fh*, *gs*, *gu*) $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s hj* Γ
then have *edenot loc s* $\in$ *dom* (*get-fh H*)
by (*metis Un-iff assms*(*1*) *domI dom-three-sum fst-conv get-fh.simps*)
then have *edenot loc s* $\in$ *dom* (*normalize* (*get-fh H*))
by (*simp add*: *dom-normalize*)
then show ¬ *aborts* (*Cwrite loc E*) (*s*, *FractionalHeap.normalize* (*get-fh H*))
by (*metis aborts-write-cases fst-eqD snd-eqD*)
qed
show *accesses* (*Cwrite loc E*) *s* $\subseteq$ *dom* (*fst* (*fh*, *gs*, *gu*)) $\wedge$ *writes* (*Cwrite loc E*) *s* $\subseteq$ *fpdom* (*fst* (*fh*, *gs*, *gu*))
using *Suc.prems*(*1*) *fpdom-def* **by** *fastforce*

fix *H hf C′ s′ h′ hj v0*

assume *asm0*: *Some H* = *Some* (*fh*, *gs*, *gu*) $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$
full-ownership (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s hj* Γ $\wedge$ *red* (*Cwrite loc E*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)
then obtain *hjf* **where** *hjf-def*: *Some hjf* = *Some hj* $\oplus$ *Some hf*
by (*metis* (*no-types*, *opaque-lifting*) *option.exhaust-sel plus.simps*(*1*) *plus-asso plus-comm*)
then have *asm00*: *Some H* = *Some* (*fh*, *gs*, *gu*) $\oplus$ *Some hjf*
by (*metis asm0 plus-asso*)
then have *get-fh hjf* (*edenot loc s*) = *None*
proof −
have *compatible-fract-heaps fh* (*get-fh hjf*)
by (*metis asm00 compatible-def compatible-eq fst-conv get-fh.elims option.discI*)
then show *?thesis* **using** *compatible-then-dom-disjoint*(*2*)[*of fh get-fh hjf*]
assms disjoint-iff-not-equal[*of dom* (*get-fh hjf*) *fpdom fh*] *not-in-dom fpdom-def mem-Collect-eq*
by *fastforce*
qed

show ∃ *h′′ H′ hj′*. *full-ownership* (*get-fh H′*) $\wedge$ *semi-consistent* Γ *v0 H′* $\wedge$ *sat-inv s′ hj′* Γ $\wedge$ *h′* = *FractionalHeap.normalize* (*get-fh H′*) $\wedge$
Some H′ = *Some h′′* $\oplus$ *Some hj′* $\oplus$ *Some hf* $\wedge$ *safe n* (*Some* Γ) *C′* (*s′*, *h′′*) {(*s*, *fh*(*edenot loc s* $\mapsto$ (*pwrite*, *edenot E s*)), *gs*, *gu*)}
proof (*rule red-write-cases*)
show *red* (*Cwrite loc E*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)

using *asm0* **by** *blast*
fix *sa h*
assume *asm1*: $(s,\ FractionalHeap.normalize\ (get\text{-}fh\ H)) = (sa,\ h)$ $C' = Cskip$ $(s',\ h') = (sa,\ h(edenot\ loc\ sa \mapsto edenot\ E\ sa))$
then obtain $s = sa$ $h' = h(edenot\ loc\ s \mapsto edenot\ E\ s)$ **by** *blast*

let $?h = (fh(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ gs,\ gu)$
have *?h ## hjf*
proof (*rule compatibleI*)
show *compatible-fract-heaps* $(get\text{-}fh\ (fh(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ gs,\ gu))\ (get\text{-}fh\ hjf)$
proof (*rule compatible-fract-heapsI*)
fix $l\ p\ p'$ **assume** *asm2*: $get\text{-}fh\ (fh(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ gs,\ gu)\ l = Some\ p \wedge get\text{-}fh\ hjf\ l = Some\ p'$
then show $pgte\ pwrite\ (padd\ (fst\ p)\ (fst\ p'))$
apply $(cases\ l = edenot\ loc\ s)$
apply (*metis Suc.prems(1) asm00 fst-conv fun-upd-same get-fh.elims option.sel plus-extract-point-fh*)
by (*metis asm00 fst-conv fun-upd-other get-fh.elims plus-extract-point-fh*)
show $snd\ p = snd\ p'$
apply $(cases\ l = edenot\ loc\ s)$
using ‹$pgte\ pwrite\ (padd\ (fst\ p)\ (fst\ p'))$› *asm2 not-pgte-charact sum-larger* **apply** *fastforce*
by (*metis (mono-tags, opaque-lifting) asm00 asm2 fst-eqD get-fh.simps map-upd-Some-unfold plus-extract-point-fh*)
qed
show $\bigwedge k.\ get\text{-}gu\ (fh(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ gs,\ gu)\ k = None \vee get\text{-}gu\ hjf\ k = None$
by (*metis asm00 compatible-def compatible-eq get-gu.simps option.discI snd-conv*)
show $\bigwedge p\ p'.\ get\text{-}gs\ (fh(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ gs,\ gu) = Some\ p \wedge get\text{-}gs\ hjf = Some\ p' \Longrightarrow pgte\ pwrite\ (padd\ (fst\ p)\ (fst\ p'))$
by (*metis asm00 compatible-def compatible-eq get-gs.simps option.discI snd-conv*)
qed
then obtain H' **where** $Some\ H' = Some\ ?h \oplus Some\ hjf$ **by** *auto*
moreover have $H' = ((get\text{-}fh\ H)(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ get\text{-}gs\ H,\ get\text{-}gu\ H)$
proof (*rule heap-ext*)
show $get\text{-}gs\ H' = get\text{-}gs\ ((get\text{-}fh\ H)(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ get\text{-}gs\ H,\ get\text{-}gu\ H)$
using *asm00 calculation*
by (*metis fst-conv get-gs.simps plus-extract(2) snd-conv*)
show $get\text{-}gu\ H' = get\text{-}gu\ ((get\text{-}fh\ H)(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ get\text{-}gs\ H,\ get\text{-}gu\ H)$
using *asm00 calculation*
by (*metis get-gu.simps plus-extract(3) snd-conv*)
show $get\text{-}fh\ H' = get\text{-}fh\ ((get\text{-}fh\ H)(edenot\ loc\ s \mapsto (pwrite,\ edenot\ E\ s)),\ get\text{-}gs\ H,\ get\text{-}gu\ H)$

proof (*rule ext*)
fix *l* **show** *get-fh* H' *l* = *get-fh* ((*get-fh H*)(*edenot loc s* $\mapsto$ (*pwrite*, *edenot E s*)), *get-gs H*, *get-gu H*) *l*
using *add-fh-update*[*of get-fh hjf edenot E s fh* (*pwrite*, *edenot E s*)]
by (*metis* ‹*get-fh hjf* (*edenot loc s*) = *None*› *add-fh-update add-get-fh asm00 calculation fst-conv get-fh.elims*)
qed
qed
moreover have *safe n* (*Some* Γ) C' (s', *?h*) {(*s*, *fh*(*edenot loc s* $\mapsto$ (*pwrite*, *edenot E s*)), *gs*, *gu*)}
using ‹*s* = *sa*› *asm1*(*2*) *asm1*(*3*) *safe-skip* **by** *fastforce*
moreover have *full-ownership* (*get-fh* H') $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H')
proof –
have *full-ownership* (*get-fh* H')
proof (*rule full-ownershipI*)
fix *l p*
assume *asm*: *get-fh* H' *l* = *Some p*
then show *fst p* = *pwrite*
proof (*cases l* = *edenot loc s*)
case *True*
then show *?thesis*
using *asm calculation*(*2*) **by** *fastforce*
next
case *False*
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *asm asm0 calculation*(*2*) *fst-eqD full-ownership-def get-fh.simps map-upd-Some-unfold*)
qed
qed
moreover have h' = *FractionalHeap.normalize* (*get-fh* H')
proof (*rule ext*)
fix *l* **show** h' *l* = *FractionalHeap.normalize* (*get-fh* H') *l*
proof(*cases l* = *edenot loc s*)
case *True*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *FractionalHeap.normalize-eq*(*2*) ‹H' = ((*get-fh H*)(*edenot loc s* $\mapsto$ (*pwrite*, *edenot E s*)), *get-gs H*, *get-gu H*)› ‹h' = *h*(*edenot loc s* $\mapsto$ *edenot E s*)› *fst-conv fun-upd-same get-fh.elims*)
next
case *False*
then have *FractionalHeap.normalize* (*get-fh* H') *l* = *FractionalHeap.normalize* (*get-fh H*) *l*
using *FractionalHeap.normalize-eq*(*2*)[*of get-fh* H' *l*]
FractionalHeap.normalize-eq(*2*)[*of get-fh H l*] ‹H' = ((*get-fh H*)(*edenot loc s* $\mapsto$ (*pwrite*, *edenot E s*)), *get-gs H*, *get-gu H*)›
fst-conv fun-upd-other[*of l edenot loc s get-fh H*] *get-fh.simps option.exhaust*
by *metis*

then show *?thesis*
using *False* ‹$h' = h(edenot\ loc\ s \mapsto edenot\ E\ s)$› *asm1*(*1*) **by** *force*
qed
qed
ultimately show *?thesis*
by *auto*
qed
moreover have *Some* H' = *Some ?h* ⊕ *Some hj* ⊕ *Some hf*
by (*metis calculation*(*1*) *hjf-def simpler-asso*)
moreover have *semi-consistent* Γ *v0* H'
proof (*rule semi-consistentI*)
show *all-guards* H'
by (*metis all-guards-def asm0 calculation*(*2*) *fst-conv get-gs.simps get-gu.simps semi-consistent-def snd-conv*)
have *view* Γ (*normalize* (*get-fh* H')) = *view* Γ (*normalize* (*get-fh H*))
proof −
have *view* Γ (*normalize* (*get-fh* H')) = *view* Γ (*normalize* (*get-fh hj*))
by (*metis asm0 assms*(*2*) *calculation*(*5*) *larger3 view-function-of-invE*)
then show *?thesis* **using** *assms*(*2*) *larger3 view-function-of-invE*
by (*metis asm0*)
qed
then show *reachable* Γ *v0* H'
by (*metis asm0 calculation*(*2*) *fst-eqD get-gs.simps get-gu.simps reachableE reachableI semi-consistent-def snd-eqD*)
qed
ultimately show ∃ h'' H' hj'.
full-ownership (*get-fh* H') ∧
semi-consistent Γ *v0* H' ∧
sat-inv s' hj' Γ ∧
h' = *FractionalHeap.normalize* (*get-fh* H') ∧
Some H' = *Some* h'' ⊕ *Some* hj' ⊕ *Some hf* ∧ *safe n* (*Some* Γ) C' (s', h'') {(*s*, *fh*(*edenot loc s* ↦ (*pwrite*, *edenot E s*)), *gs*, *gu*)}
using ‹*s* = *sa*› *asm0 asm1*(*2*) *asm1*(*3*) **by** *blast*
qed
qed
qed (*simp*)

lemma *safe-write*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*
assumes *fh* (*edenot loc s*) = *Some* (*pwrite*, *v*)
and ⋀Γ. Δ = *Some* Γ ⟹ *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*)) }
apply (*cases* Δ)
using *safe-write-None safe-write-Some assms* **by** *blast*+

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

assumes $\bigwedge\Gamma.\ \Delta = \mathit{Some}\ \Gamma \Longrightarrow \mathit{view\text{-}function\text{-}of\text{-}inv}\ \Gamma$
and $v \notin \mathit{fvE}\ \mathit{loc}$
shows $\mathit{hoare\text{-}triple\text{-}valid}\ \Delta\ (\mathit{Exists}\ v\ (\mathit{PointsTo}\ \mathit{loc}\ \mathit{pwrite}\ (\mathit{Evar}\ v)))\ (\mathit{Cwrite}\ \mathit{loc}\ E)\ (\mathit{PointsTo}\ \mathit{loc}\ \mathit{pwrite}\ E)$
proof $(\mathit{rule}\ \mathit{hoare\text{-}triple\text{-}validI})$

define $\Sigma :: \mathit{store} \times ('i,\ 'a)\ \mathit{heap} \Rightarrow (\mathit{store} \times ('i,\ 'a)\ \mathit{heap})\ \mathit{set}$ **where**
$\Sigma = (\lambda(s,\ h).\ \{\ (s,\ ((\mathit{get\text{-}fh}\ h)(\mathit{edenot}\ \mathit{loc}\ s \mapsto (\mathit{pwrite},\ \mathit{edenot}\ E\ s)),\ \mathit{get\text{-}gs}\ h,\ \mathit{get\text{-}gu}\ h))\ \})$

show $\bigwedge s\ h\ n.\ (s,\ h),\ (s,\ h) \models \mathit{Exists}\ v\ (\mathit{PointsTo}\ \mathit{loc}\ \mathit{pwrite}\ (\mathit{Evar}\ v)) \Longrightarrow \mathit{safe}\ n\ \Delta\ (\mathit{Cwrite}\ \mathit{loc}\ E)\ (s,\ h)\ (\Sigma\ (s,\ h))$
proof $-$
fix $s\ h\ n$ **assume** $(s,\ h :: ('i,\ 'a)\ \mathit{heap}),\ (s,\ h) \models \mathit{Exists}\ v\ (\mathit{PointsTo}\ \mathit{loc}\ \mathit{pwrite}\ (\mathit{Evar}\ v))$
then obtain vv **where** $(s(v := \mathit{vv}),\ h),\ (s(v := \mathit{vv}),\ h) \models \mathit{PointsTo}\ \mathit{loc}\ \mathit{pwrite}\ (\mathit{Evar}\ v)$
by $(\mathit{meson}\ \mathit{hyper\text{-}sat.simps}(6)\ \mathit{hyper\text{-}sat.simps}(7))$
then have $\mathit{get\text{-}fh}\ h\ (\mathit{edenot}\ \mathit{loc}\ (s(v := \mathit{vv}))) = \mathit{Some}\ (\mathit{pwrite},\ \mathit{vv})$
by *simp*
then have $\mathit{get\text{-}fh}\ h\ (\mathit{edenot}\ \mathit{loc}\ s) = \mathit{Some}\ (\mathit{pwrite},\ \mathit{vv})$
using $\mathit{assms}(2)$ **by** *auto*
then show $\mathit{safe}\ n\ \Delta\ (\mathit{Cwrite}\ \mathit{loc}\ E)\ (s,\ h)\ (\Sigma\ (s,\ h))$
by $(\mathit{metis}\ (\mathit{mono\text{-}tags},\ \mathit{lifting})\ \Sigma\text{-}\mathit{def}\ \mathit{assms}(1)\ \mathit{decompose\text{-}heap\text{-}triple}\ \mathit{old.prod.case}\ \mathit{safe\text{-}write})$
qed
fix $s1\ h1\ s2\ h2$
assume $(s1,\ h1 :: ('i,\ 'a)\ \mathit{heap}),\ (s2,\ h2) \models \mathit{Exists}\ v\ (\mathit{PointsTo}\ \mathit{loc}\ \mathit{pwrite}\ (\mathit{Evar}\ v))$
then obtain $v1\ v2$ **where** $\mathit{get\text{-}fh}\ h1\ (\mathit{edenot}\ \mathit{loc}\ s1) = \mathit{Some}\ (\mathit{pwrite},\ v1)\ \mathit{get\text{-}fh}\ h2\ (\mathit{edenot}\ \mathit{loc}\ s2) = \mathit{Some}\ (\mathit{pwrite},\ v2)$
using $\mathit{assms}(2)$ **by** *auto*

show $\mathit{pair\text{-}sat}\ (\mathit{case}\ (s1,\ h1)\ \mathit{of}\ (s,\ h) \Rightarrow \{(s,\ (\mathit{get\text{-}fh}\ h)(\mathit{edenot}\ \mathit{loc}\ s \mapsto (\mathit{pwrite},\ \mathit{edenot}\ E\ s)),\ \mathit{get\text{-}gs}\ h,\ \mathit{get\text{-}gu}\ h)\})$
$(\mathit{case}\ (s2,\ h2)\ \mathit{of}\ (s,\ h) \Rightarrow \{(s,\ (\mathit{get\text{-}fh}\ h)(\mathit{edenot}\ \mathit{loc}\ s \mapsto (\mathit{pwrite},\ \mathit{edenot}\ E\ s)),\ \mathit{get\text{-}gs}\ h,\ \mathit{get\text{-}gu}\ h)\})\ (\mathit{PointsTo}\ \mathit{loc}\ \mathit{pwrite}\ E)$
proof $(\mathit{rule}\ \mathit{pair\text{-}satI})$
fix $s1'\ h1'\ s2'\ h2'$
assume $\mathit{asm0}$: $(s1',\ h1') \in (\mathit{case}\ (s1,\ h1)\ \mathit{of}\ (s,\ h) \Rightarrow \{(s,\ (\mathit{get\text{-}fh}\ h)(\mathit{edenot}\ \mathit{loc}\ s \mapsto (\mathit{pwrite},\ \mathit{edenot}\ E\ s)),\ \mathit{get\text{-}gs}\ h,\ \mathit{get\text{-}gu}\ h)\}) \wedge$
$(s2',\ h2') \in (\mathit{case}\ (s2,\ h2)\ \mathit{of}\ (s,\ h) \Rightarrow \{(s,\ (\mathit{get\text{-}fh}\ h)(\mathit{edenot}\ \mathit{loc}\ s \mapsto (\mathit{pwrite},\ \mathit{edenot}\ E\ s)),\ \mathit{get\text{-}gs}\ h,\ \mathit{get\text{-}gu}\ h)\})$
then show $(s1',\ h1'),\ (s2',\ h2') \models \mathit{PointsTo}\ \mathit{loc}\ \mathit{pwrite}\ E$
using $\langle(s1,\ h1),\ (s2,\ h2) \models \mathit{Exists}\ v\ (\mathit{PointsTo}\ \mathit{loc}\ \mathit{pwrite}\ (\mathit{Evar}\ v))\rangle\ \mathit{assms}(2)$ **by** *auto*
qed
qed

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* $\mapsto$ (π, *v*)], *gs*, *gu*))
{ (*s*(*x* := *v*), ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*)) }
proof (*induct n*)
case (*Suc n*)
show *?case*
proof (*rule safeNoneI*)

show *no-abort* (*None* :: ('*i*, '*a*, *nat*) *cont*) (*Cread x E*) *s* ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*)
proof (*rule no-abortNoneI*)
fix *hf H*
assume *asm0*: *Some H* = *Some* ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*) $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H*
then have *edenot E s* $\in$ *dom* (*get-fh H*)
by (*metis Un-iff dom-eq-singleton-conv dom-sum-two fst-eqD get-fh.elims insert-iff*)
then have *edenot E s* $\in$ *dom* (*FractionalHeap.normalize* (*get-fh H*))
by (*simp add*: *dom-normalize*)
then show $\neg$ *aborts* (*Cread x E*) (*s*, *FractionalHeap.normalize* (*get-fh H*))
by (*metis aborts-read-cases fst-eqD snd-eqD*)
qed

show *accesses* (*Cread x E*) *s* $\subseteq$ *dom* (*fst* ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*)) $\wedge$ *writes* (*Cread x E*) *s* $\subseteq$ *fpdom* (*fst* ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*))
by *simp*

fix *H hf* C' s' h'
assume *asm0*: *Some H* = *Some* ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*) $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H* $\wedge$ *red* (*Cread x E*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) C' (s', h')

let *?S* = { (*s*(*x* := *v*), ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*)) }

show $\exists$ h'' H'.
full-ownership (*get-fh* H') $\wedge$
no-guard H' $\wedge$
h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$
Some H' = *Some* h'' $\oplus$ *Some hf* $\wedge$ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (s', h'') *?S*
proof (*rule red-read-cases*)

show *red* (*Cread x E*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) C' (s', h')
using *asm0* **by** *blast*

fix *sa h va*
assume $(s,\ \textit{FractionalHeap.normalize}\ (\textit{get-fh}\ H)) = (sa,\ h)$ $C' = \textit{Cskip}$ $(s',\ h') = (sa(x := va),\ h)$
$h\ (\textit{edenot}\ E\ sa) = \textit{Some}\ va$
then have $s = sa$
by *force*
then have $va = v$
proof −
have $\exists\pi'.\ \textit{get-fh}\ H\ (\textit{edenot}\ E\ s) = \textit{Some}\ (\pi',\ v)$
proof (*rule one-value-sum-same*)
show $\textit{Some}\ H = \textit{Some}\ ([\textit{edenot}\ E\ s \mapsto (\pi,\ v)],\ gs,\ gu) \oplus \textit{Some}\ hf$
using *asm0* **by** *fastforce*
qed (*simp*)
then show *?thesis*
by (*metis FractionalHeap.normalize-eq(2) Pair-inject* ‹$(s,\ \textit{FractionalHeap.normalize}\ (\textit{get-fh}\ H)) = (sa,\ h)$› ‹$h\ (\textit{edenot}\ E\ sa) = \textit{Some}\ va$› *option.sel*)
qed

then have $\textit{safe}\ n\ (\textit{None} :: ('i,\ 'a,\ nat)\ \textit{cont})\ C'\ (s',\ ([\textit{edenot}\ E\ s \mapsto (\pi,\ v)],\ gs,\ gu))\ ?S$
using ‹$(s',\ h') = (sa(x := va),\ h)$› ‹$C' = \textit{Cskip}$› ‹$s = sa$› *safe-skip* **by** *fastforce*

then show $\exists h''\ H'.$
$\textit{full-ownership}\ (\textit{get-fh}\ H') \wedge \textit{no-guard}\ H' \wedge$
$h' = \textit{FractionalHeap.normalize}\ (\textit{get-fh}\ H') \wedge \textit{Some}\ H' = \textit{Some}\ h'' \oplus \textit{Some}\ hf \wedge \textit{safe}\ n\ (\textit{None} :: ('i,\ 'a,\ nat)\ \textit{cont})\ C'\ (s',\ h'')\ \{(s(x := v),\ [\textit{edenot}\ E\ s \mapsto (\pi,\ v)],\ gs,\ gu)\}$
using ‹$(s',\ h') = (sa(x := va),\ h)$› ‹$(s,\ \textit{FractionalHeap.normalize}\ (\textit{get-fh}\ H)) = (sa,\ h)$› *asm0* **by** *blast*
qed
qed (*simp*)
qed (*simp*)

lemma *safe-read-Some*:
assumes *view-function-of-inv* Γ
and $x \notin \textit{fvA}\ (\textit{invariant}\ \Gamma)$
shows $\textit{safe}\ n\ (\textit{Some}\ \Gamma)\ (\textit{Cread}\ x\ E)\ (s,\ ([\textit{edenot}\ E\ s \mapsto (\pi,\ v)],\ gs,\ gu))\ \{\ (s(x := v),\ ([\textit{edenot}\ E\ s \mapsto (\pi,\ v)],\ gs,\ gu))\ \}$
proof (*induct n*)
case (*Suc n*)
show *?case*
proof (*rule safeSomeI*)

show $\textit{no-abort}\ (\textit{Some}\ \Gamma)\ (\textit{Cread}\ x\ E)\ s\ ([\textit{edenot}\ E\ s \mapsto (\pi,\ v)],\ gs,\ gu)$
proof (*rule no-abortSomeI*)
fix *hf H hj v0*
assume *asm0*: $\textit{Some}\ H = \textit{Some}\ ([\textit{edenot}\ E\ s \mapsto (\pi,\ v)],\ gs,\ gu) \oplus \textit{Some}\ hj \oplus \textit{Some}\ hf \wedge \textit{full-ownership}\ (\textit{get-fh}\ H) \wedge \textit{semi-consistent}\ \Gamma\ v0\ H \wedge \textit{sat-inv}\ s\ hj\ \Gamma$

then obtain *hjf* **where** *Some H* = *Some* ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*) $\oplus$ *Some hjf*
by (*metis* (*no-types*, *lifting*) *plus.simps*(*2*) *plus.simps*(*3*) *plus-asso*)
then have *edenot E s* $\in$ *dom* (*get-fh H*)
by (*metis Un-iff dom-eq-singleton-conv dom-sum-two fst-eqD get-fh.elims insert-iff*)
then have *edenot E s* $\in$ *dom* (*FractionalHeap.normalize* (*get-fh H*))
by (*simp add*: *dom-normalize*)
then show $\neg$ *aborts* (*Cread x E*) (*s*, *FractionalHeap.normalize* (*get-fh H*))
by (*metis aborts-read-cases fst-eqD snd-eqD*)
qed
show *accesses* (*Cread x E*) *s* $\subseteq$ *dom* (*fst* ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*)) $\wedge$ *writes* (*Cread x E*) *s* $\subseteq$ *fpdom* (*fst* ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*))
by *simp*

fix *H hf C′ s′ h′ hj v0*
assume *asm0*: *Some H* = *Some* ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*) $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$
full-ownership (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s hj* Γ $\wedge$ *red* (*Cread x E*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)
then obtain *hjf* **where** *Some hjf* = *Some hj* $\oplus$ *Some hf*
using *compatible-last-two* **by** (*metis plus.simps*(*3*) *plus-asso*)
then have *Some H* = *Some* ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*) $\oplus$ *Some hjf*
by (*metis asm0 plus-asso*)

let *?S* = { (*s*(*x* := *v*), ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*)) }

show $\exists$ *h″ H′ hj′*. *full-ownership* (*get-fh H′*) $\wedge$ *semi-consistent* Γ *v0 H′* $\wedge$ *sat-inv s′ hj′* Γ $\wedge$ *h′* = *FractionalHeap.normalize* (*get-fh H′*) $\wedge$
Some H′ = *Some h″* $\oplus$ *Some hj′* $\oplus$ *Some hf* $\wedge$ *safe n* (*Some* Γ) *C′* (*s′*, *h″*) {(*s*(*x* := *v*), [*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*)}
proof (*rule red-read-cases*)

show *red* (*Cread x E*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)
using *asm0* **by** *blast*
fix *sa h va*
assume (*s*, *FractionalHeap.normalize* (*get-fh H*)) = (*sa*, *h*) *C′* = *Cskip* (*s′*, *h′*) = (*sa*(*x* := *va*), *h*)
h (*edenot E sa*) = *Some va*
then have *s* = *sa*
by *force*
then have *va* = *v*
proof −
have $\exists$ π'. *get-fh H* (*edenot E s*) = *Some* (π', *v*)
proof (*rule one-value-sum-same*)
show *Some H* = *Some* ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*) $\oplus$ *Some hjf*
by (*simp add*: ‹*Some H* = *Some* ([*edenot E s* $\mapsto$ (π, *v*)], *gs*, *gu*) $\oplus$ *Some hjf*›)
qed (*simp*)

then show *?thesis*
by (*metis FractionalHeap.normalize-eq*(*2*) *Pair-inject* ‹(*s*, *FractionalHeap.normalize* (*get-fh H*)) = (*sa*, *h*)› ‹*h* (*edenot E sa*) = *Some va*› *option.sel*)
qed

then have *safe n* (*Some* Γ) *C′* (*s′*, ([*edenot E s* ↦ (π, *v*)], *gs*, *gu*)) *?S*
using ‹(*s′*, *h′*) = (*sa*(*x* := *va*), *h*)› ‹*C′* = *Cskip*› ‹*s* = *sa*› *safe-skip* **by** *fastforce*
moreover have *sat-inv s′ hj* Γ
by (*metis* ‹(*s′*, *h′*) = (*sa*(*x* := *va*), *h*)› ‹*s* = *sa*› *agrees-update asm0 assms*(*2*) *prod.inject sat-inv-agrees*)
ultimately show ∃ *h′′ H′ hj′*.
full-ownership (*get-fh H′*) ∧ *semi-consistent* Γ *v0 H′* ∧ *sat-inv s′ hj′* Γ ∧ *h′* = *FractionalHeap.normalize* (*get-fh H′*) ∧
Some H′ = *Some h′′* ⊕ *Some hj′* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C′* (*s′*, *h′′*) {(*s*(*x* := *v*), [*edenot E s* ↦ (π, *v*)], *gs*, *gu*)}
using ‹(*s′*, *h′*) = (*sa*(*x* := *va*), *h*)› ‹(*s*, *FractionalHeap.normalize* (*get-fh H*)) = (*sa*, *h*)› *asm0* **by** *blast*
qed
qed (*simp*)
qed (*simp*)

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*)) }
apply (*cases* Δ)
using *safe-read-None safe-read-Some assms* **by** *blast+*

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 (*rule hoare-triple-validI*)

define Σ :: *store* × (*′i*, *′a*) *heap* ⇒ (*store* × (*′i*, *′a*) *heap*) *set* **where**
Σ = (λ(*s*, *h*). { (*s*(*x* := *edenot e s*), ([*edenot E s* ↦ (π, *edenot e s*)], *get-gs h*, *get-gu h*)) })

show ⋀*s h n*. (*s*, *h*), (*s*, *h*) ⊨ *PointsTo E* π *e* ⟹ *safe n* Δ (*Cread x E*) (*s*, *h*) (Σ (*s*, *h*))
proof −
fix *s h n*
assume (*s*, *h* :: (*′i*, *′a*) *heap*), (*s*, *h*) ⊨ *PointsTo E* π *e*
then have *get-fh h* = [*edenot E s* ↦ (π, *edenot e s*)]
using *sat-points-to* **by** *blast*

then have $h = ([edenot\ E\ s \mapsto (\pi,\ edenot\ e\ s)],\ get\text{-}gs\ h,\ get\text{-}gu\ h)$
by (*metis decompose-heap-triple*)
then have *safe n* Δ (*Cread x E*) (s, ([*edenot E s* $\mapsto$ (π, *edenot e s*)], *get-gs h*, *get-gu h*))
{ ($s(x := edenot\ e\ s)$, ([*edenot E s* $\mapsto$ (π, *edenot e s*)], *get-gs h*, *get-gu h*)) }
using *assms safe-read* **by** *blast*
then show *safe n* Δ (*Cread x E*) (s, h) (Σ (s, h))
using Σ-*def* ‹h = ([*edenot E s* $\mapsto$ (π, *edenot e s*)], *get-gs h*, *get-gu h*)› **by** *auto*
qed

fix *s1 h1 s2 h2*
assume (*s1*, *h1* :: ('*i*, '*a*) *heap*), (*s2*, *h2*) $\models$ *PointsTo E* π *e*

show *pair-sat* (*case* (*s1*, *h1*) *of* (s, h) $\Rightarrow$ {($s(x := edenot\ e\ s)$, [*edenot E s* $\mapsto$ (π, *edenot e s*)], *get-gs h*, *get-gu h*)})
(*case* (*s2*, *h2*) *of* (s, h) $\Rightarrow$ {($s(x := edenot\ e\ s)$, [*edenot E s* $\mapsto$ (π, *edenot e s*)], *get-gs h*, *get-gu h*)}) (*And* (*PointsTo E* π *e*) (*Bool* (*Beq* (*Evar x*) *e*)))
proof (*rule pair-satI*)
fix *s1' h1' s2' h2'*
assume *asm0*: (*s1'*, *h1'*) $\in$ (*case* (*s1*, *h1*) *of* (s, h) $\Rightarrow$ {($s(x := edenot\ e\ s)$, [*edenot E s* $\mapsto$ (π, *edenot e s*)], *get-gs h*, *get-gu h*)}) $\wedge$
(*s2'*, *h2'*) $\in$ (*case* (*s2*, *h2*) *of* (s, h) $\Rightarrow$ {($s(x := edenot\ e\ s)$, [*edenot E s* $\mapsto$ (π, *edenot e s*)], *get-gs h*, *get-gu h*)})
then obtain *s1'* = *s1*(x := *edenot e s1*) *h1'* = ([*edenot E s1* $\mapsto$ (π, *edenot e s1*)], *get-gs h1*, *get-gu h1*)
s2' = *s2*(x := *edenot e s2*) *h2'* = ([*edenot E s2* $\mapsto$ (π, *edenot e s2*)], *get-gs h2*, *get-gu h2*)
by *force*
then show (*s1'*, *h1'*), (*s2'*, *h2'*) $\models$ *And* (*PointsTo E* π *e*) (*Bool* (*Beq* (*Evar x*) *e*))
using *assms*(*2*) **by** *auto*
qed
qed

4.4.6 Share

lemma *share-no-abort*:
assumes *no-abort* (*Some* Γ) *C s* (h :: ('*i*, '*a*) *heap*)
and *Some* (h' :: ('*i*, '*a*) *heap*) = *Some h* $\oplus$ *Some hj*
and *sat-inv s hj* Γ
and *get-gs h* = *Some* (*pwrite*, *sargs*)
and $\bigwedge k$. *get-gu h k* = *Some* (*uargs k*)
and *reachable-value* (*saction* Γ) (*uaction* Γ) *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 (*rule no-abortI*)
show $\bigwedge H$ *hf hj v0* Γ.
None = *Some* Γ $\wedge$

Some H = *Some* (*remove-guards h'*) $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s hj* $\Gamma \Longrightarrow$
$\neg$ *aborts C* (*s*, *FractionalHeap.normalize* (*get-fh H*)) **by** *blast*

fix *hf H* :: ('*i*, '*a*) *heap*
assume *asm0*: *Some H* = *Some* (*remove-guards h'*) $\oplus$ *Some hf* $\wedge$ *None* = *None* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H*

have *compatible h' hf*
proof (*rule compatibleI*)
show *compatible-fract-heaps* (*get-fh h'*) (*get-fh hf*)
by (*metis asm0 compatible-def compatible-eq fst-eqD get-fh.simps option.distinct*(*1*) *remove-guards-def*)
show $\bigwedge$*k*. *get-gu h' k* = *None* $\vee$ *get-gu hf k* = *None*
by (*metis asm0 no-guard-def no-guard-then-smaller-same plus-comm*)
fix *p p'* **assume** *get-gs h'* = *Some p* $\wedge$ *get-gs hf* = *Some p'*
then show *pgte pwrite* (*padd* (*fst p*) (*fst p'*))
by (*metis asm0 no-guard-def no-guard-then-smaller-same option.distinct*(*1*) *plus-comm*)
qed
then obtain *H'* **where** *Some H'* = *Some h'* $\oplus$ *Some hf*
by *simp*
then have *get-fh H'* = *get-fh H*
by (*metis asm0 fst-eqD get-fh.elims option.discI remove-guards-def option.sel plus.simps*(*3*))

have $\neg$ *aborts C* (*s*, *FractionalHeap.normalize* (*get-fh H'*))
proof (*rule no-abortE*(*2*))
show *no-abort* (*Some* Γ) *C s h*
using *assms* **by** *blast*
show *Some* Γ = *Some* Γ **by** *blast*
show *full-ownership* (*get-fh H'*)
using ‹*get-fh H'* = *get-fh H*› *asm0* **by** *presburger*
show *semi-consistent* Γ *v0 H'*
proof (*rule semi-consistentI*)
show *all-guards H'*
by (*metis* ‹*Some H'* = *Some h'* $\oplus$ *Some hf*› *all-guards-def all-guards-same assms*(*2*) *assms*(*4*) *assms*(*5*) *option.discI*)

have *view* Γ (*normalize* (*get-fh hj*)) = *view* Γ (*normalize* (*get-fh H'*))
using *assms*(*7*)
proof (*rule view-function-of-invE*)
show *H'* $\succeq$ *hj*
using *larger-trans*
by (*simp add*: ‹*Some H'* = *Some h'* $\oplus$ *Some hf*› *assms*(*2*) *larger3*)
show *sat-inv s hj* Γ
by (*simp add*: *assms*(*3*))
qed

show *reachable* Γ *v0 H′*
proof (*rule reachableI*)
fix *sargs′ uargs′*
assume *asm1*: *get-gs H′* = *Some* (*pwrite*, *sargs′*) ∧ (∀ *k*. *get-gu H′ k* = *Some* (*uargs′ k*))
then have *sargs* = *sargs′*
by (*metis* ‹*Some H′* = *Some h′* ⊕ *Some hf*› *assms*(*2*) *assms*(*4*) *full-sguard-sum-same option.inject snd-conv*)
moreover have *uargs* = *uargs′*
proof (*rule ext*)

fix *k*
show *uargs k* = *uargs′ k*
using *full-uguard-sum-same*[*of h′ k* - *H′ hf*]
by (*metis* ‹*Some H′* = *Some h′* ⊕ *Some hf*› *asm1 assms*(*2*) *assms*(*5*) *full-uguard-sum-same option.inject*)
qed
ultimately show *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs′ uargs′* (*view* Γ (*FractionalHeap.normalize* (*get-fh H′*)))
using ‹*view* Γ (*FractionalHeap.normalize* (*get-fh hj*)) = *view* Γ (*FractionalHeap.normalize* (*get-fh H′*))› *assms*(*6*) **by** *presburger*
qed
qed
show *Some H′* = *Some h* ⊕ *Some hj* ⊕ *Some hf*
using ‹*Some H′* = *Some h′* ⊕ *Some hf*› *assms*(*2*) **by** *presburger*
show *sat-inv s hj* Γ
by (*simp add*: *assms*(*3*))
qed

then show ¬ *aborts C* (*s*, *FractionalHeap.normalize* (*get-fh H*))
using ‹*get-fh H′* = *get-fh H*› **by** *auto*
qed

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
and *bounded h′*
shows *safe n* (*None* :: (*′i*, *′a*, *nat*) *cont*) *C* (*s*, *remove-guards h′*) (*S-after-share S* Γ *v0*)
using *assms*
proof (*induct n arbitrary*: *C s h h′ hj*)
case (*Suc n*)

let *?S′* = *S-after-share S* Γ *v0*

have *is-in-s′*: $\bigwedge$*h hj h′*. *Some h′* = *Some h* ⊕ *Some hj* ∧ (*s*, *h*) ∈ *S* ∧ *sat-inv s hj* Γ ∧ *semi-consistent* Γ *v0 h′* ⟹ (*s*, *remove-guards h′*) ∈ *?S′*
proof −
fix *h hj h′* **assume** *Some h′* = *Some h* ⊕ *Some hj* ∧ (*s*, *h*) ∈ *S* ∧ *sat-inv s hj* Γ ∧ *semi-consistent* Γ *v0 h′*
then show (*s*, *remove-guards h′*) ∈ *?S′*
using *S-after-share-def*[*of S* Γ *v0*] *mem-Collect-eq* **by** *blast*
qed
show *?case*
proof (*rule safeNoneI*)

show *C* = *Cskip* ⟹ (*s*, *remove-guards h′*) ∈ *?S′*
proof −
assume *C* = *Cskip*
show (*s*, *remove-guards h′*) ∈ *?S′*
proof (*rule is-in-s′*)
show *Some h′* = *Some h* ⊕ *Some hj* ∧ (*s*, *h*) ∈ *S* ∧ *sat-inv s hj* Γ ∧ *semi-consistent* Γ *v0 h′*
using *Suc.prems* ‹*C* = *Cskip*› *safeSomeE*(*1*) *sat-inv-def* **by** *blast*
qed
qed

obtain *sargs uargs* **where** *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 (*FractionalHeap.normalize* (*get-fh h′*)))
by (*meson Suc.prems*(*4*) *semi-consistentE*)
show *no-abort None C s* (*remove-guards h′*)
proof (*rule share-no-abort*)
show *no-abort* (*Some* Γ) *C s h*
using *Suc.prems*(*1*) *safeSomeE*(*2*) **by** *blast*
show *Some h′* = *Some h* ⊕ *Some hj*
using *Suc.prems*(*2*) **by** *blast*
show *sat-inv s hj* Γ
using *Suc.prems*(*3*) **by** *auto*
show *get-gs h* = *Some* (*pwrite*, *sargs*)
by (*metis Suc.prems*(*2*) ‹*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 (*FractionalHeap.normalize* (*get-fh h′*)))› ‹*sat-inv s hj* Γ› *no-guard-remove*(*1*) *sat-inv-def*)
show $\bigwedge$*k*. *get-gu h k* = *Some* (*uargs k*)
by (*metis Suc.prems*(*2*) ‹*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 (*FractionalHeap.normalize* (*get-fh h′*)))› ‹*sat-inv s hj* Γ› *no-guard-remove*(*2*) *sat-inv-def*)
show *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs uargs* (*view* Γ (*FractionalHeap.normalize*

$(get\text{-}fh\ hj)))$
by $(metis\ Suc.prems(2)\ Suc.prems(3)\ \langle get\text{-}gs\ h' = Some\ (pwrite,\ sargs) \wedge (\forall k.\ get\text{-}gu\ h'\ k = Some\ (uargs\ k)) \wedge reachable\text{-}value\ (saction\ \Gamma)\ (uaction\ \Gamma)\ v0\ sargs\ uargs\ (view\ \Gamma\ (FractionalHeap.normalize\ (get\text{-}fh\ h')))\rangle\ assms(5)\ larger\text{-}def\ plus\text{-}comm\ view\text{-}function\text{-}of\text{-}invE)$
show $view\text{-}function\text{-}of\text{-}inv\ \Gamma$
by $(simp\ add\colon assms(5))$
qed

have $accesses\ C\ s \subseteq dom\ (fst\ h) \wedge writes\ C\ s \subseteq fpdom\ (fst\ h)$
using $Suc.prems(1)$ **by** $force$
moreover have $dom\ (fst\ h) \subseteq dom\ (fst\ h')$
by $(metis\ Suc.prems(2)\ addition\text{-}smaller\text{-}domain\ get\text{-}fh.simps)$
moreover have $fpdom\ (fst\ h) \subseteq fpdom\ (fst\ h')$
by $(simp\ add\colon Suc.prems(2)\ Suc(7)\ fpdom\text{-}inclusion)$
ultimately show $accesses\ C\ s \subseteq dom\ (fst\ (remove\text{-}guards\ h')) \wedge writes\ C\ s \subseteq fpdom\ (fst\ (remove\text{-}guards\ h'))$
unfolding $remove\text{-}guards\text{-}def$ **by** $force$

fix $H\ hf\ C'\ s'\ h'a$
assume $asm0\colon Some\ H = Some\ (remove\text{-}guards\ h') \oplus Some\ hf \wedge full\text{-}ownership\ (get\text{-}fh\ H) \wedge no\text{-}guard\ H \wedge red\ C\ (s,\ FractionalHeap.normalize\ (get\text{-}fh\ H))\ C'\ (s',\ h'a)$

have $compatible\ h'\ hf$
proof $(rule\ compatibleI)$
show $compatible\text{-}fract\text{-}heaps\ (get\text{-}fh\ h')\ (get\text{-}fh\ hf)$
by $(metis\ asm0\ compatible\text{-}def\ compatible\text{-}eq\ fst\text{-}eqD\ get\text{-}fh.simps\ option.distinct(1)\ remove\text{-}guards\text{-}def)$
show $\bigwedge k.\ get\text{-}gu\ h'\ k = None \vee get\text{-}gu\ hf\ k = None$
by $(metis\ asm0\ no\text{-}guard\text{-}def\ no\text{-}guard\text{-}then\text{-}smaller\text{-}same\ plus\text{-}comm)$
fix $p\ p'$ **assume** $get\text{-}gs\ h' = Some\ p \wedge get\text{-}gs\ hf = Some\ p'$
then show $pgte\ pwrite\ (padd\ (fst\ p)\ (fst\ p'))$
by $(metis\ asm0\ no\text{-}guard\text{-}def\ no\text{-}guard\text{-}then\text{-}smaller\text{-}same\ option.distinct(1)\ plus\text{-}comm)$
qed
then obtain Hg **where** $Some\ Hg = Some\ h' \oplus Some\ hf$
by $simp$
then have $get\text{-}fh\ Hg = get\text{-}fh\ H$
by $(metis\ asm0\ fst\text{-}eqD\ get\text{-}fh.elims\ option.discI\ remove\text{-}guards\text{-}def\ option.sel\ plus.simps(3))$

have $\exists h''\ H'\ hj'.$
$full\text{-}ownership\ (get\text{-}fh\ H') \wedge$
$semi\text{-}consistent\ \Gamma\ v0\ H' \wedge$
$sat\text{-}inv\ s'\ hj'\ \Gamma \wedge h'a = FractionalHeap.normalize\ (get\text{-}fh\ H') \wedge Some\ H' = Some\ h'' \oplus Some\ hj' \oplus Some\ hf \wedge safe\ n\ (Some\ \Gamma)\ C'\ (s',\ h'')\ S$
using $Suc(2)$

proof (*rule safeSomeE*(*3*)[*of n* Γ *C s h S Hg hj hf v0 C′ s′ h′a*])
show *Some Hg* = *Some h* ⊕ *Some hj* ⊕ *Some hf*
by (*simp add*: *Suc.prems*(*2*) ‹*Some Hg* = *Some h′* ⊕ *Some hf*›)
show *full-ownership* (*get-fh Hg*)
using ‹*get-fh Hg* = *get-fh H*› *asm0* **by** *presburger*
show *sat-inv s hj* Γ
by (*simp add*: *Suc.prems*(*3*))
show *red C* (*s*, *FractionalHeap.normalize* (*get-fh Hg*)) *C′* (*s′*, *h′a*)
using ‹*get-fh Hg* = *get-fh H*› *asm0* **by** *presburger*
show *semi-consistent* Γ *v0 Hg*
proof (*rule semi-consistentI*)
show *all-guards Hg*
by (*meson Suc.prems*(*4*) ‹*Some Hg* = *Some h′* ⊕ *Some hf*› *all-guards-same semi-consistent-def*)
have *view* Γ (*normalize* (*get-fh hj*)) = *view* Γ (*normalize* (*get-fh Hg*))
using *assms*(*5*)
proof (*rule view-function-of-invE*)
show *Hg* ⪰ *hj*
using *larger-trans*
using ‹*Some Hg* = *Some h* ⊕ *Some hj* ⊕ *Some hf*› *larger3* **by** *blast*
show *sat-inv s hj* Γ
by (*simp add*: ‹*sat-inv s hj* Γ›)
qed
show *reachable* Γ *v0 Hg*
proof (*rule reachableI*)
fix *sargs′ uargs′*
assume *asm1*: *get-gs Hg* = *Some* (*pwrite*, *sargs′*) ∧ (∀ *k*. *get-gu Hg k* = *Some* (*uargs′ k*))
then have *sargs* = *sargs′*
by (*metis Pair-inject* ‹*Some Hg* = *Some h′* ⊕ *Some hf*› ‹*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 (*FractionalHeap.normalize* (*get-fh h′*)))› *full-sguard-sum-same option.inject*)
moreover have *uargs* = *uargs′*
proof (*rule ext*)
fix *k*
show *uargs k* = *uargs′ k*
by (*metis* ‹*Some Hg* = *Some h′* ⊕ *Some hf*› ‹*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 (*FractionalHeap.normalize* (*get-fh h′*)))› *asm1 full-uguard-sum-same option.inject*)
qed
ultimately show *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs′ uargs′* (*view* Γ (*FractionalHeap.normalize* (*get-fh Hg*)))
by (*metis Suc.prems*(*2*) ‹*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 (*FractionalHeap.normalize* (*get-fh h′*)))› ‹*sat-inv s hj* Γ› ‹*view* Γ (*FractionalHeap.normalize* (*get-fh hj*)) = *view* Γ (*FractionalHeap.normalize* (*get-fh Hg*))› *assms*(*5*) *larger-def plus-comm view-function-of-invE*)

qed
qed
qed
then obtain h'' H' hj' **where** *asm1*: *full-ownership* (*get-fh* H') $\wedge$ *semi-consistent* Γ *v0* H' $\wedge$
sat-inv s' hj' Γ $\wedge$ $h'a$ = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some* hj' $\oplus$ *Some* *hf* $\wedge$ *safe* *n* (*Some* Γ) C' (s', h'') *S*
by *blast*
obtain hj'' **where** *Some* hj'' = *Some* h'' $\oplus$ *Some* hj'
by (*metis asm1 not-Some-eq plus.simps*(*1*))
moreover obtain *sargs'* *uargs'* **where** *new-guards-def*:
get-gs H' = *Some* (*pwrite*, *sargs'*) $\wedge$ ($\forall$ *k*. *get-gu* H' *k* = *Some* (*uargs'* *k*)) $\wedge$ *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs' uargs'* (*view* Γ (*FractionalHeap.normalize* (*get-fh* H')))
by (*meson asm1 semi-consistentE*)

have *safe* *n* (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (s', *remove-guards* hj'') *?S'*
proof (*rule Suc*(*1*)[*of* C' s' h'' hj'' hj'])

show *safe* *n* (*Some* Γ) C' (s', h'') *S*
using *asm1* **by** *blast*
show *Some* hj'' = *Some* h'' $\oplus$ *Some* hj'
using ‹*Some* hj'' = *Some* h'' $\oplus$ *Some* hj'› **by** *blast*
show *sat-inv* s' hj' Γ
using *asm1* **by** *fastforce*

have *no-guard hf*
by (*metis asm0 no-guard-then-smaller-same plus-comm*)
moreover have *no-guard* hj'
using ‹*sat-inv* s' hj' Γ› *sat-inv-def* **by** *blast*

have *view* Γ (*normalize* (*get-fh* hj')) = *view* Γ (*normalize* (*get-fh* H'))
using *assms*(*5*)
proof (*rule view-function-of-invE*)
show $H' \succeq hj'$
using *larger-trans*
using *asm1 larger3* **by** *blast*
show *sat-inv* s' hj' Γ
by (*simp add*: *asm1*)
qed

obtain *uargs'* *sargs'* **where** *args'*: *get-gs* H' = *Some* (*pwrite*, *sargs'*) $\wedge$ ($\forall$ *k*. *get-gu* H' *k* = *Some* (*uargs'* *k*)) $\wedge$ *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs'* *uargs'*
(*view* Γ (*FractionalHeap.normalize* (*get-fh* H')))
using *semi-consistentE*[*of* Γ *v0* H'] *asm1*
by *blast*

then have *get-gs hj''* = *Some* (*pwrite*, *sargs'*) ∧ (∀ *k*. *get-gu hj'' k* = *Some* (*uargs' k*))
by (*metis* ‹*Some hj''* = *Some h''* ⊕ *Some hj'*› *asm1 calculation no-guard-remove*(*1*) *no-guard-remove*(*2*))

show *semi-consistent* Γ *v0 hj''*
proof (*rule semi-consistentI*)

show *all-guards hj''*
by (*metis* ‹*get-gs hj''* = *Some* (*pwrite*, *sargs'*) ∧ (∀ *k*. *get-gu hj'' k* = *Some* (*uargs' k*))› *all-guards-def option.discI*)
have *view* Γ (*FractionalHeap.normalize* (*get-fh H'*)) = *view* Γ (*FractionalHeap.normalize* (*get-fh hj''*))
by (*metis* ‹*Some hj''* = *Some h''* ⊕ *Some hj'*› ‹*view* Γ (*FractionalHeap.normalize* (*get-fh hj'*)) = *view* Γ (*FractionalHeap.normalize* (*get-fh H'*))› *asm1 assms*(*5*) *larger-def plus-comm view-function-of-invE*)
then show *reachable* Γ *v0 hj''*
by (*metis* ‹*get-gs hj''* = *Some* (*pwrite*, *sargs'*) ∧ (∀ *k*. *get-gu hj'' k* = *Some* (*uargs' k*))› *args' asm1 ext get-fh.simps new-guards-def option.sel reachable-def snd-conv*)
qed
show *view-function-of-inv* Γ
by (*simp add*: *assms*(*5*))
show *bounded hj''*
proof (*rule bounded-smaller*)
show *bounded H'*
by (*metis asm1 full-ownership-then-bounded get-fh.simps*)
show *H'* ⪰ *hj''*
by (*metis* ‹*Some hj''* = *Some h''* ⊕ *Some hj'*› *asm1 larger-def*)
qed
qed

let *?h''* = *remove-guards hj''*
have *hj''* ## *hf*
by (*metis asm1 calculation option.simps*(*3*) *plus.simps*(*3*))
then obtain *H''* **where** *Some H''* = *Some ?h''* ⊕ *Some hf*
by (*simp add*: *remove-guards-smaller smaller-more-compatible*)

then have *get-fh H''* = *get-fh H'*
by (*metis asm1 calculation equiv-sum-get-fh get-fh-remove-guards*)
moreover have *no-guard H''*
by (*metis* ‹*Some H''* = *Some* (*remove-guards hj''*) ⊕ *Some hf*› *asm0 no-guard-remove-guards no-guard-then-smaller-same plus-comm sum-of-no-guards*)

ultimately show ∃ *h'' H'*.
full-ownership (*get-fh H'*) ∧
no-guard H' ∧ *h'a* = *FractionalHeap.normalize* (*get-fh H'*) ∧ *Some H'* = *Some h''* ⊕ *Some hf* ∧ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C'* (*s'*, *h''*) *?S'*
by (*metis* ‹*Some H''* = *Some* (*remove-guards hj''*) ⊕ *Some hf*› ‹*safe n None*

C' $(s'$, *remove-guards* $hj'')$ *?S'*› *asm1*)
  qed
qed (*simp*)

definition *no-need-guards* **where**
  no-need-guards $A \longleftrightarrow (\forall$ *s1 h1 s2 h2*. (*s1*, *h1*), (*s2*, *h2*) $\models A \longrightarrow$ (*s1*, *remove-guards h1*), (*s2*, *remove-guards h2*) $\models A$)

lemma *has-guard-then-safe-none*:
  assumes $\neg$ *no-guard h*
    and C = *Cskip* $\Longrightarrow (s, h) \in S$
    and *accesses C s* $\subseteq$ *dom* (*fst h*) $\wedge$ *writes C s* $\subseteq$ *fpdom* (*fst h*)
  shows *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) C (s, h) S
proof (*induct n*)
  case (*Suc n*)
  show *?case*
  proof (*rule safeNoneI*)
    show C = *Cskip* $\Longrightarrow (s, h) \in S$
      by (*simp add*: *assms*(*2*))
    show *no-abort None C s h*
      using *assms*(*1*) *no-abortNoneI no-guard-then-smaller-same* **by** *blast*
    show *accesses C s* $\subseteq$ *dom* (*fst h*) $\wedge$ *writes C s* $\subseteq$ *fpdom* (*fst h*)
      using *assms*(*3*) **by** *blast*
    show $\bigwedge$*H hf* C' s' h'.
      Some H = *Some h* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H* $\wedge$ *red C* (*s*, *FractionalHeap.normalize* (*get-fh H*)) C' $(s', h') \Longrightarrow$
      $\exists h''$ H'.
        full-ownership (*get-fh* H') $\wedge$ *no-guard* H' $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some hf* $\wedge$ *safe n None* C' (s', h'') S
      using *assms*(*1*) *no-guard-then-smaller-same* **by** *blast*
  qed
qed (*simp*)

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* $\wedge$ *precise J*
    and *wf-indexed-precondition upre* $\wedge$ *wf-precondition spre*
    and $x \notin$ *fvA J*
    and *no-guard-assertion* (*Star P* (*LowView* ($\alpha \circ f$) *J x*))

shows *hoare-triple-valid* (*None* :: (*'i*, *'a*, *nat*) *cont*) (*Star P* (*LowView* ($\alpha \circ f$) *J x*)) *C* (*Star Q* (*LowView* ($\alpha \circ f$) *J x*))
proof −
let *?P* = *Star P EmptyFullGuards*
let *?Q* = *Star Q* (*And* (*PreSharedGuards* (*Abs-precondition spre*)) (*PreUniqueGuards* (*Abs-indexed-precondition upre*)))

obtain Σ **where** *asm0*: $\bigwedge\sigma$ *n*. σ, σ $\models$ *Star P EmptyFullGuards* $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe n* (*Some* Γ) *C* σ (Σ σ)
$\bigwedge\sigma\ \sigma'$. σ, σ' $\models$ *Star P EmptyFullGuards* $\Longrightarrow$ *pair-sat* (Σ σ) (Σ σ') (*Star Q* (*And* (*PreSharedGuards* (*Abs-precondition spre*)) (*PreUniqueGuards* (*Abs-indexed-precondition upre*))))
using *hoare-triple-validE*[*of Some* Γ *?P C ?Q*] *assms*(*3*) **by** *blast*

Steps: 1) Remove the hj and add empty-guards 2) Apply sigma 3) Remove the guards and add hj, using S-after-share

define *input-*Σ **where** *input-*Σ = ($\lambda\sigma$. { (*fst* σ, *add-empty-guards hp*) |*hp hj*. *Some* (*snd* σ) = *Some hp* $\oplus$ *Some hj* $\wedge$
(*fst* σ, *hp*), (*fst* σ, *hp*) $\models$ *P* $\wedge$ *sat-inv* (*fst* σ) *hj* Γ})

define Σ' **where** Σ' = ($\lambda\sigma$. $\bigcup$*p* $\in$ *input-*Σ σ. *S-after-share* (Σ *p*) Γ (*f* (*normalize* (*get-fh* (*snd* σ)))))

show *?thesis*
proof (*rule hoare-triple-validI-bounded*)
show $\bigwedge$*s h n*. (*s*, *h*), (*s*, *h*) $\models$ *Star P* (*LowView* ($\alpha \circ f$) *J x*) $\Longrightarrow$ *bounded h* $\Longrightarrow$ *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C* (*s*, *h*) (Σ' (*s*, *h*))
proof −
fix *s h n* **assume** *asm1*: (*s*, *h*), (*s*, *h*) $\models$ *Star P* (*LowView* ($\alpha \circ f$) *J x*) *bounded h*
then obtain *hp hj* **where** *no-guard h Some h* = *Some hp* $\oplus$ *Some hj* (*s*, *hp*), (*s*, *hp*) $\models$ *P*
(*s*, *hj*), (*s*, *hj*) $\models$ *LowView* ($\alpha \circ f$) *J x*
by (*meson always-sat-refl assms*(*8*) *hyper-sat.simps*(*4*) *no-guard-assertion-def*)
then have *sat-inv s hj* Γ
by (*metis LowViewE assms*(*1*) *assms*(*7*) *no-guard-then-smaller-same plus-comm sat-inv-def select-convs*(*5*))
then have (*s*, *add-empty-guards hp*) $\in$ *input-*Σ (*s*, *h*)
using ‹(*s*, *hp*), (*s*, *hp*) $\models$ *P*› ‹*Some h* = *Some hp* $\oplus$ *Some hj*› *input-*Σ*-def* **by** *force*

let *?v0* = *f* (*normalize* (*get-fh h*))
let *?p* = (*s*, *add-empty-guards hp*)

have *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C* (*s*, *remove-guards* (*add-empty-guards h*)) (*S-after-share* (Σ *?p*) Γ *?v0*)
proof (*rule share-lemma*)
show *safe n* (*Some* Γ) *C ?p* (Σ *?p*)
proof (*rule asm0*(*1*))

show $(s,\ \textit{add-empty-guards hp}),\ (s,\ \textit{add-empty-guards hp}) \models \textit{Star P EmptyFullGuards}$
using ‹$(s,\ hp),\ (s,\ hp) \models P$› ‹$\textit{Some h} = \textit{Some hp} \oplus \textit{Some hj}$› ‹*no-guard h*› *no-guard-and-sat-p-empty-guards no-guard-then-smaller-same* **by** *blast*
show *bounded* (*snd* (*s*, *add-empty-guards hp*))
unfolding *bounded-def add-empty-guards-def* **apply** *simp*
by (*metis* ‹$\textit{Some h} = \textit{Some hp} \oplus \textit{Some hj}$› *asm1*(*2*) *boundedE bounded-smaller-sum fst-eqD*)
qed
show $\textit{Some}\ (\textit{add-empty-guards h}) = \textit{Some}\ (\textit{add-empty-guards hp}) \oplus \textit{Some hj}$
using ‹$\textit{Some h} = \textit{Some hp} \oplus \textit{Some hj}$› ‹*no-guard h*› *no-guard-add-empty-guards-sum* **by** *blast*
show *sat-inv s hj* Γ
using ‹*sat-inv s hj* Γ› **by** *auto*
show *view-function-of-inv* Γ
by (*simp add*: *assms*(*4*))
show *semi-consistent* Γ (*f* (*FractionalHeap.normalize* (*get-fh h*))) (*add-empty-guards h*)
by (*metis* ‹*no-guard h*› *assms*(*1*) *select-convs*(*1*) *semi-consistent-empty-no-guard-initial-value*)
show *bounded* (*add-empty-guards h*)
unfolding *bounded-def add-empty-guards-def* **apply** *simp*
by (*metis asm1*(*2*) *boundedE fst-eqD*)
qed
moreover have $(\textit{S-after-share}\ (\Sigma\ ?p)\ \Gamma\ ?v0) \subseteq \Sigma'\ (s,\ h)$
using Σ'*-def* ‹$(s,\ \textit{add-empty-guards hp}) \in \textit{input-}\Sigma\ (s,\ h)$› **by** *auto*
ultimately show *safe n* $(\textit{None} :: ('i,\ 'a,\ \textit{nat})\ \textit{cont})\ C\ (s,\ h)\ (\Sigma'\ (s,\ h))$
by (*metis* ‹*no-guard h*› *no-guards-remove-same safe-larger-set*)
qed

fix *s1 h1 s2 h2*
assume $(s1,\ h1),\ (s2,\ h2) \models \textit{Star P}\ (\textit{LowView}\ (\alpha \circ f)\ J\ x)$
then obtain *hp1 hj1 hp2 hj2* **where** *asm1*: $\textit{Some h1} = \textit{Some hp1} \oplus \textit{Some hj1}$
$\textit{Some h2} = \textit{Some hp2} \oplus \textit{Some hj2}$ $(s1,\ hp1),\ (s2,\ hp2) \models P$ *no-guard h1 no-guard h2*
$(s1,\ hj1),\ (s2,\ hj2) \models \textit{LowView}\ (\alpha \circ f)\ J\ x$
using *assms*(*8*) *hyper-sat.simps*(*4*) *no-guard-assertion-def* **by** *blast*
then obtain $(s1,\ hj1),\ (s2,\ hj2) \models J$ $\alpha\ (f\ (\textit{normalize}\ (\textit{get-fh hj1}))) = \alpha\ (f\ (\textit{normalize}\ (\textit{get-fh hj2})))$
by (*metis LowViewE assms*(*7*) *comp-apply*)

show *pair-sat* $(\Sigma'\ (s1,\ h1))\ (\Sigma'\ (s2,\ h2))\ (\textit{Star Q}\ (\textit{LowView}\ (\alpha \circ f)\ J\ x))$
proof (*rule pair-satI*)
fix *s1' h1' s2' h2'*
assume *asm2*: $(s1',\ h1') \in \Sigma'\ (s1,\ h1) \land (s2',\ h2') \in \Sigma'\ (s2,\ h2)$
then obtain *p1 p2* **where** *p-assms*: $p1 \in \textit{input-}\Sigma\ (s1,\ h1)$ $p2 \in \textit{input-}\Sigma\ (s2,\ h2)$
$(s1',\ h1') \in \textit{S-after-share}\ (\Sigma\ p1)\ \Gamma\ (f\ (\textit{normalize}\ (\textit{get-fh h1})))$
$(s2',\ h2') \in \textit{S-after-share}\ (\Sigma\ p2)\ \Gamma\ (f\ (\textit{normalize}\ (\textit{get-fh h2})))$

using Σ'-*def* **by** *force*
moreover have *pair-sat* (Σ *p1*) (Σ *p2*) (*Star Q* (*And* (*PreSharedGuards* (*Abs-precondition spre*)) (*PreUniqueGuards* (*Abs-indexed-precondition upre*))))
proof (*rule asm0*(*2*))
obtain *hj1' hj2' hp1' hp2'* **where** *snd p1* = *add-empty-guards hp1' snd p2* = *add-empty-guards hp2'*
Some h1 = *Some hp1'* $\oplus$ *Some hj1' Some h2* = *Some hp2'* $\oplus$ *Some hj2' sat-inv s1 hj1'* Γ *sat-inv s2 hj2'* Γ
fst p1 = *s1 fst p2* = *s2*
using *p-assms*(*1*) *p-assms*(*2*) *input*-Σ-*def* **by** *auto*
moreover have *hj1* = *hj1'* $\wedge$ *hj2* = *hj2'*
proof (*rule preciseE*)
show *precise J*
by (*simp add*: *assms*(*5*))
show *h1* $\succeq$ *hj1'* $\wedge$ *h1* $\succeq$ *hj1* $\wedge$ *h2* $\succeq$ *hj2'* $\wedge$ *h2* $\succeq$ *hj2*
by (*metis asm1*(*1*) *asm1*(*2*) *calculation*(*3*) *calculation*(*4*) *larger-def plus-comm*)
show (*s1*, *hj1'*), (*s2*, *hj2'*) $\models$ *J* $\wedge$ (*s1*, *hj1*), (*s2*, *hj2*) $\models$ *J*
by (*metis* ‹(*s1*, *hj1*), (*s2*, *hj2*) $\models$ *J*› *assms*(*1*) *assms*(*5*) *calculation*(*5*) *calculation*(*6*) *sat-inv-def select-convs*(*5*) *unaryE*)
qed
then have *hp1* = *hp1'* $\wedge$ *hp2* = *hp2'*
using *addition-cancellative asm1*(*1*) *asm1*(*2*) *calculation*(*3*) *calculation*(*4*) **by** *blast*
then show *p1*, *p2* $\models$ *Star P EmptyFullGuards*
using *no-guard-and-sat-p-empty-guards*[*of fst p1 snd p1 fst p2 snd p2 P*]
by (*metis asm1*(*3*) *asm1*(*4*) *asm1*(*5*) *calculation*(*1*) *calculation*(*2*) *calculation*(*3*) *calculation*(*4*) *calculation*(*7*) *calculation*(*8*) *no-guard-and-sat-p-empty-guards no-guard-then-smaller-same prod.exhaust-sel*)
qed

let *?v1* = *f* (*normalize* (*get-fh h1*))
let *?v2* = *f* (*normalize* (*get-fh h2*))

obtain *hj1' hg1 H1 hj2' hg2 H2* **where** *asm3*: *h1'* = *remove-guards H1 semi-consistent* Γ *?v1 H1*
Some H1 = *Some hg1* $\oplus$ *Some hj1'* (*s1'*, *hg1*) $\in$ Σ *p1 sat-inv s1' hj1'* Γ
h2' = *remove-guards H2 semi-consistent* Γ *?v2 H2*
Some H2 = *Some hg2* $\oplus$ *Some hj2'* (*s2'*, *hg2*) $\in$ Σ *p2 sat-inv s2' hj2'* Γ
using *p-assms*(*3*) *S-after-share-def*[*of* Σ *p1* Γ *?v1*] *p-assms*(*4*) *S-after-share-def*[*of* Σ *p2* Γ *?v2*] **by** *blast*

then have (*s1'*, *hg1*), (*s2'*, *hg2*) $\models$ *Star Q* (*And* (*PreSharedGuards* (*Abs-precondition spre*)) (*PreUniqueGuards* (*Abs-indexed-precondition upre*)))
using ‹*pair-sat* (Σ *p1*) (Σ *p2*) (*Star Q* (*And* (*PreSharedGuards* (*Abs-precondition spre*)) (*PreUniqueGuards* (*Abs-indexed-precondition upre*))))› *pair-satE* **by** *blast*
then obtain *q1 g1 q2 g2* **where** *Some hg1* = *Some q1* $\oplus$ *Some g1 Some hg2* = *Some q2* $\oplus$ *Some g2*

(s1', q1), (s2', q2) ⊨ Q (s1', g1), (s2', g2) ⊨ PreSharedGuards (Abs-precondition spre) (s1', g1), (s2', g2) ⊨ PreUniqueGuards (Abs-indexed-precondition upre)
by *(meson hyper-sat.simps(3) hyper-sat.simps(4))*
moreover have *Rep-precondition (Abs-precondition spre) = spre ∧ Rep-indexed-precondition (Abs-indexed-precondition upre) = upre*
by *(simp add: assms(6) wf-indexed-precondition-rep-prec wf-precondition-rep-prec)*
ultimately obtain *sargs1 sargs2* **where**
get-gs g1 = Some (pwrite, sargs1) get-gs g2 = Some (pwrite, sargs2) PRE-shared-simpler spre sargs1 sargs2
get-fh g1 = Map.empty get-fh g2 = Map.empty
by *auto*
moreover obtain *uargs1 uargs2* **where**
unique-facts: ⋀k. get-gu g1 k = Some (uargs1 k) ∧ get-gu g2 k = Some (uargs2 k) ∧ PRE-unique (upre k) (uargs1 k) (uargs2 k)
using *sat-PreUniqueE[OF ‹(s1', g1), (s2', g2) ⊨ PreUniqueGuards (Abs-indexed-precondition upre)›]*
by *(metis ‹Rep-precondition (Abs-precondition spre) = spre ∧ Rep-indexed-precondition (Abs-indexed-precondition upre) = upre›)*
moreover obtain *get-gs H1 = Some (pwrite, sargs1) ⋀k. get-gu H1 k = Some (uargs1 k)*
by *(metis (no-types, opaque-lifting) ‹Some hg1 = Some q1 ⊕ Some g1› asm3(3) calculation(1) calculation(6) full-sguard-sum-same full-uguard-sum-same plus-comm)*
then have *reach1: reachable-value sact uact ?v1 sargs1 uargs1 (f (normalize (get-fh H1)))*
by *(metis asm3(2) assms(1) reachableE select-convs(1) select-convs(3) select-convs(4) semi-consistent-def)*
moreover obtain *get-gs H2 = Some (pwrite, sargs2) ⋀k. get-gu H2 k = Some (uargs2 k)*
by *(metis (no-types, lifting) ‹Some hg2 = Some q2 ⊕ Some g2› asm3(8) calculation(2) calculation(6) full-sguard-sum-same full-uguard-sum-same plus-comm)*
then have *reach2: reachable-value sact uact ?v2 sargs2 uargs2 (f (normalize (get-fh H2)))*
by *(metis asm3(7) assms(1) reachableE semi-consistent-def simps(1) simps(3) simps(4))*
moreover have *α (f (normalize (get-fh h1))) = α (f (normalize (get-fh hj1)))*
using *view-function-of-invE[of Γ s1 hj1 h1]*
by *(metis ‹(s1, hj1), (s2, hj2) ⊨ J› always-sat-refl asm1(1) asm1(4) assms(1) assms(4) larger-def no-guard-then-smaller-same plus-comm sat-inv-def select-convs(1) select-convs(5))*
moreover have *α (f (normalize (get-fh h2))) = α (f (normalize (get-fh hj2)))*
using *view-function-of-invE[of Γ s2 hj2 h2]*
by *(metis ‹(s1, hj1), (s2, hj2) ⊨ J› always-sat-refl asm1(2) asm1(5) assms(1) assms(4) larger-def no-guard-then-smaller-same plus-comm sat-comm sat-inv-def select-convs(1) select-convs(5))*
ultimately have *low-abstract-view: α (f (FractionalHeap.normalize (get-fh H1))) = α (f (FractionalHeap.normalize (get-fh H2)))*
using *reach1 reach2 main-result[of sact uact ?v1 sargs1 uargs1 f (normalize (get-fh H1)) ?v2 sargs2 uargs2 f (normalize (get-fh H2)) spre upre α]*

using ‹α (f (*FractionalHeap.normalize* (*get-fh hj1*))) = α (f (*FractionalHeap.normalize* (*get-fh hj2*)))› *assms*(*2*) **by** *presburger*
moreover have α (f (*normalize* (*get-fh H1*))) = α (f (*normalize* (*get-fh hj1′*)))
using *view-function-of-invE*[*of* Γ *s1′ hj1′ H1*]
by (*metis asm3*(*3*) *asm3*(*5*) *assms*(*1*) *assms*(*4*) *larger-def plus-comm select-convs*(*1*))
moreover have α (f (*normalize* (*get-fh H2*))) = α (f (*normalize* (*get-fh hj2′*)))
using *view-function-of-invE*[*of* Γ *s2′ hj2′ H2*]
by (*metis asm3*(*10*) *asm3*(*8*) *assms*(*1*) *assms*(*4*) *larger-def plus-comm select-convs*(*1*))
moreover have (*s1′*, *hj1′*), (*s2′*, *hj2′*) $\models$ *J*
by (*metis asm3*(*10*) *asm3*(*5*) *assms*(*1*) *assms*(*5*) *sat-inv-def select-convs*(*5*) *unaryE*)
ultimately have (*s1′*, *hj1′*), (*s2′*, *hj2′*) $\models$ *LowView* ($\alpha \circ f$) *J x*
by (*simp add*: *LowViewI assms*(*7*))
moreover have *Some h1′* = *Some q1* $\oplus$ *Some hj1′*
proof −
have *Some h1′* = *Some* (*remove-guards hg1*) $\oplus$ *Some* (*remove-guards hj1′*)
using *asm3*(*1*) *asm3*(*3*) *remove-guards-sum* **by** *blast*
moreover have *remove-guards hg1* = *remove-guards q1*
by (*metis* ‹*Some hg1* = *Some q1* $\oplus$ *Some g1*› ‹*get-fh g1* = *Map.empty*› *get-fh-remove-guards no-guard-and-no-heap no-guard-remove-guards no-guards-remove remove-guards-sum*)
moreover have *remove-guards hj1′* = *hj1′*
by (*metis asm3*(*5*) *no-guards-remove sat-inv-def*)
ultimately show *?thesis*
by (*metis* ‹*Some hg1* = *Some q1* $\oplus$ *Some g1*› ‹*get-gs g1* = *Some* (*pwrite*, *sargs1*)› *unique-facts all-guards-def full-guard-comp-then-no no-guards-remove option.distinct*(*1*) *plus.simps*(*3*) *plus-comm*)
qed
moreover have *Some h2′* = *Some q2* $\oplus$ *Some hj2′*
proof −
have *Some h2′* = *Some* (*remove-guards hg2*) $\oplus$ *Some* (*remove-guards hj2′*)
using *asm3*(*6*) *asm3*(*8*) *remove-guards-sum* **by** *blast*
moreover have *remove-guards hg2* = *remove-guards q2*
by (*metis* ‹*Some hg2* = *Some q2* $\oplus$ *Some g2*› ‹*get-fh g2* = *Map.empty*› *get-fh-remove-guards no-guard-and-no-heap no-guard-remove-guards no-guards-remove remove-guards-sum*)
moreover have *remove-guards hj2′* = *hj2′*
by (*metis asm3*(*10*) *no-guards-remove sat-inv-def*)
ultimately show *?thesis*
by (*metis* ‹*Some hg2* = *Some q2* $\oplus$ *Some g2*› ‹*get-gs g2* = *Some* (*pwrite*, *sargs2*)› *unique-facts all-guards-def full-guard-comp-then-no no-guards-remove option.distinct*(*1*) *plus.simps*(*3*) *plus-comm*)
qed
ultimately show (*s1′*, *h1′*), (*s2′*, *h2′*) $\models$ *Star Q* (*LowView* ($\alpha \circ f$) *J x*)
by (*meson LowViewI* ‹(*s1′*, *q1*), (*s2′*, *q2*) $\models$ *Q*› *assms*(*7*) *hyper-sat.simps*(*9*)

hyper-sat.simps(*4*))
  qed
 qed
qed

4.4.7 Atomic

lemma *red-rtrans-induct*:
 assumes *red-rtrans* $C\ \sigma\ C'\ \sigma'$
  and $\bigwedge C\ \sigma.\ P\ C\ \sigma\ C\ \sigma$
  and $\bigwedge C\ \sigma\ C'\ \sigma'\ C''\ \sigma''.$ *red* $C\ \sigma\ C'\ \sigma' \Longrightarrow$ *red-rtrans* $C'\ \sigma'\ C''\ \sigma'' \Longrightarrow P\ C'\ \sigma'\ C''\ \sigma'' \Longrightarrow P\ C\ \sigma\ C''\ \sigma''$
 shows $P\ C\ \sigma\ C'\ \sigma'$
 using *red-red-rtrans.inducts*[*of* - - - - λ- - - -. *True P*] *assms* **by** *auto*

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* σ*2* ∧ (*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*))
 using *assms*
proof (*induction arbitrary*: *s1 H1 H h rule*: *red-rtrans-induct*[*of C1* σ*1 C2* σ*2*])
 case *1*
 then show *?case* **by** (*simp add*: *assms*(*1*))
next
 case (*2 C* σ)
 then have ¬ *aborts C* (*s1*, *FractionalHeap.normalize* (*get-fh H*))
  using *no-abortE*(*1*) *safe.simps*(*2*) **by** *blast*
 then have ¬ *aborts C* σ
  by (*metis 2.prems*(*2*) *2.prems*(*5*) *denormalize-properties*(*3*))
 moreover have *safe* (*Suc 1*) (*None* :: (*'i*, *'a*, *nat*) *cont*) *C* (*s1*, *h*) *S*
  using *2.prems*(*4*) **by** *blast*
 then have *C* = *Cskip* ⟹ (*s2*, *h*) ∈ *S*
  by (*metis 2.prems*(*2*) *2.prems*(*3*) *Pair-inject safeNoneE*(*1*))
 then have *C* = *Cskip* ⟹ *Some H* = *Some h* ⊕ *Some hf* ∧ *H2* = *FractionalHeap.normalize* (*get-fh H*) ∧ *no-guard H* ∧ *full-ownership* (*get-fh H*) ∧ (*s2*, *h*) ∈ *S*
   by (*metis 2.prems*(*3*) *2.prems*(*2*) *2.prems*(*6*) *2.prems*(*5*) *2.prems*(*7*) *denormalize-properties*(*3*) *old.prod.inject*)
 ultimately show *?case*
  by *blast*
next
 case (*3 C* σ C' σ' C'' σ'')

obtain *s0 H0* **where** $\sigma' = (s0, H0)$ **using** *prod.exhaust-sel* **by** *blast*

have *safe* (*Suc 0*) (*None* :: ('*i*, '*a*, *nat*) *cont*) *C* (*s1*, *h*) *S*
using *3.prems*(*4*) **by** *force*
then have $\exists$ *h''* *H'*. *full-ownership* (*get-fh H'*) $\wedge$ *no-guard H'* $\wedge$ *H0* = *FractionalHeap.normalize* (*get-fh H'*) $\wedge$ *Some H'* = *Some h''* $\oplus$ *Some hf* $\wedge$ *safe 0* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C'* (*s0*, *h''*) *S*
proof (*rule safeNoneE*(*3*)[*of 0 C s1 h S H hf C' s0 H0*])
show *Some H* = *Some h* $\oplus$ *Some hf* **using** *3.prems*(*6*) **by** *blast*
show *full-ownership* (*get-fh H*) **using** *3.prems*(*7*) **by** *blast*
show *no-guard H* **using** *3.prems*(*7*) **by** *auto*
show *red C* (*s1*, *FractionalHeap.normalize* (*get-fh H*)) *C'* (*s0*, *H0*)
by (*metis 3.hyps*(*1*) *3.prems*(*2*) *3.prems*(*5*) ‹$\sigma' = (s0, H0)$› *denormalize-properties*(*3*))
qed
then obtain *h0 H0'* **where**
r1: *full-ownership* (*get-fh H0'*) $\wedge$ *no-guard H0'* $\wedge$ *H0* = *FractionalHeap.normalize* (*get-fh H0'*) $\wedge$ *Some H0'* = *Some h0* $\oplus$ *Some hf* $\wedge$ *safe 0* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C'* (*s0*, *h0*) *S*
by *blast*
then have *Some* (*denormalize H0*) = *Some h0* $\oplus$ *Some hf*
by (*metis denormalize-properties*(*4*))
have *ih*:
$\neg$ *aborts C''* σ'' $\wedge$ (*C''* = *Cskip* $\longrightarrow$
($\exists$ *h1 H'*. *Some H'* = *Some h1* $\oplus$ *Some hf* $\wedge$ *H2* = *FractionalHeap.normalize* (*get-fh H'*) $\wedge$ *no-guard H'* $\wedge$ *full-ownership* (*get-fh H'*) $\wedge$ (*s2*, *h1*) $\in$ *S*))
proof (*rule 3*(*3*)[*of s0 H0 h0 H0'*])
show $\sigma' = (s0, H0)$ **by** (*simp add*: ‹$\sigma' = (s0, H0)$›)
show $\sigma'' = (s2, H2)$
by (*simp add*: *3.prems*(*3*))
show *H0'* = *denormalize H0* **by** (*metis denormalize-properties*(*4*) *r1*)
show *Some H0'* = *Some h0* $\oplus$ *Some hf* **using** *r1* **by** *blast*
show *full-ownership* (*get-fh H0'*) $\wedge$ *no-guard H0'* **using** *r1* **by** *blast*
show *red-rtrans C'* σ' *C''* σ''
by (*simp add*: *3.hyps*(*2*))

fix *n*
have *safe* (*Suc n*) (*None* :: ('*i*, '*a*, *nat*) *cont*) *C* (*s1*, *h*) *S*
using *3.prems*(*4*) **by** *force*

then have $\exists$ *h''* *H'*. *full-ownership* (*get-fh H'*) $\wedge$ *no-guard H'* $\wedge$ *H0* = *FractionalHeap.normalize* (*get-fh H'*) $\wedge$ *Some H'* = *Some h''* $\oplus$ *Some hf* $\wedge$ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C'* (*s0*, *h''*) *S*
proof (*rule safeNoneE*(*3*)[*of n C s1 h S H hf C' s0 H0*])
show *Some H* = *Some h* $\oplus$ *Some hf* **using** *3.prems*(*6*) **by** *blast*
show *full-ownership* (*get-fh H*) **using** *3.prems*(*7*) **by** *blast*
show *no-guard H* **using** *3.prems*(*7*) **by** *auto*
show *red C* (*s1*, *FractionalHeap.normalize* (*get-fh H*)) *C'* (*s0*, *H0*)
by (*metis 3.hyps*(*1*) *3.prems*(*2*) *3.prems*(*5*) ‹$\sigma' = (s0, H0)$› *denormal-*

ize-properties(*3*))
  qed
  then obtain *h3 H3′* **where**
  r2: *full-ownership* (*get-fh H3′*) ∧ *no-guard H3′* ∧ *H0* = *FractionalHeap.normalize* (*get-fh H3′*) ∧ *Some H3′* = *Some h3* ⊕ *Some hf* ∧ *safe n* (*None* :: (*′i*, *′a*, *nat*) *cont*) *C′* (*s0*, *h3*) *S*
    by *blast*
  then have *h3* = *h0*
  by (*metis* ‹*Some* (*denormalize H0*) = *Some h0* ⊕ *Some hf*› *addition-cancellative denormalize-properties*(*4*))
  moreover have *H3′* = *H0′*
    by (*metis* ‹*Some H0′* = *Some h0* ⊕ *Some hf*› *calculation option.inject r2*)
  ultimately show *safe n* (*None* :: (*′i*, *′a*, *nat*) *cont*) *C′* (*s0*, *h0*) *S* **using** *r2* **by** *blast*
 qed
 then show *?case* **by** *blast*
qed

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 −
 let *?J* = *View f J* (λ*s*. *s x*)
 let *?J′* = *View f J* (λ*s*. *uact index* (*s x*) (*map-to-arg* (*s uarg*)))

let *?pre-l* = ($\lambda s.$ *map-to-list* (*s l*))
let *?G* = *UniqueGuard index ?pre-l*
let *?l* = $\lambda s.$ *map-to-arg* (*s uarg*) # *map-to-list* (*s l*)
let *?G'* = *UniqueGuard index ?l*

have *unaries*: *unary ?J* $\wedge$ *unary ?J'*
by (*simp add*: *assms*(*3*) *unary-inv-then-view*)
moreover have *precises*: *precise ?J* $\wedge$ *precise ?J'*
by (*simp add*: *assms*(*3*) *precise-inv-then-view*)

obtain Σ **where** *asm0*: $\bigwedge n\ \sigma.\ \sigma, \sigma \models$ *Star P ?J* $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C* σ ($\Sigma\ \sigma$)
$\bigwedge \sigma\ \sigma'.\ \sigma, \sigma' \models$ *Star P ?J* $\Longrightarrow$ *pair-sat* ($\Sigma\ \sigma$) ($\Sigma\ \sigma'$) (*Star Q ?J'*)
using *assms*(*2*) *hoare-triple-valid-def* **by** *blast*

define *start* **where** *start* = ($\lambda\sigma$. { (*s*, *h*) |*s h hj*. *agrees* ($-$ {*x*}) (*fst* σ) *s* $\wedge$ *Some h* = *Some* (*remove-guards* (*snd* σ)) $\oplus$ *Some hj* $\wedge$ (*s*, *hj*), (*s*, *hj*) $\models$ *?J*})
define *end-qj* **where** *end-qj* = ($\lambda\sigma.\ \bigcup\sigma' \in$ *start* σ. $\Sigma\ \sigma'$)
define Σ' **where** Σ' = ($\lambda\sigma$. { (*s*, *add-uguard-to-no-guard index hq* (*?l s*)) |*s hq h hj*. (*s*, *h*) $\in$ *end-qj* σ $\wedge$ *Some h* = *Some hq* $\oplus$ *Some hj* $\wedge$ (*s*, *hj*), (*s*, *hj*) $\models$ *?J'* })

let *?*Σ' = $\lambda\sigma$. *close-var* ($\Sigma'\ \sigma$) *x*

show *hoare-triple-valid* (*Some* Γ) (*Star P ?G*) (*Catomic C*) (*Star Q ?G'*)
proof (*rule hoare-triple-validI-bounded*)
show $\bigwedge s\ h\ s'\ h'$. (*s*, *h*), (*s'*, *h'*) $\models$ *Star P ?G* $\Longrightarrow$ *pair-sat* (*?*Σ' (*s*, *h*)) (*?*Σ' (*s'*, *h'*)) (*Star Q ?G'*)
proof $-$
fix *s1 h1 s2 h2*
assume *asm1*: (*s1*, *h1*), (*s2*, *h2*) $\models$ *Star P ?G*
then obtain *p1 p2 g1 g2* **where** *r0*: *Some h1* = *Some p1* $\oplus$ *Some g1*
Some h2 = *Some p2* $\oplus$ *Some g2*
(*s1*, *p1*), (*s2*, *p2*) $\models$ *P* (*s1*, *g1*), (*s2*, *g2*) $\models$ *?G*
using *hyper-sat.simps*(*4*) **by** *auto*
then obtain *remove-guards h1* = *p1* *remove-guards h2* = *p2*
by (*meson assms*(*10*) *hyper-sat.simps*(*13*) *no-guard-and-no-heap no-guard-assertion-def*)

have *pair-sat* (Σ' (*s1*, *h1*)) (Σ' (*s2*, *h2*)) (*Star Q ?G'*)
proof (*rule pair-satI*)
fix *s1' hqg1 s2' hqg2* $\sigma2'$
assume *asm2*: (*s1'*, *hqg1*) $\in \Sigma'$ (*s1*, *h1*) $\wedge$ (*s2'*, *hqg2*) $\in \Sigma'$ (*s2*, *h2*)
then obtain *h1' hj1' h2' hj2' hq1 hq2* **where** *r*: (*s1'*, *h1'*) $\in$ *end-qj* (*s1*, *h1*) *Some h1'* = *Some hq1* $\oplus$ *Some hj1'*
(*s1'*, *hj1'*), (*s1'*, *hj1'*) $\models$ *?J'* (*s2'*, *h2'*) $\in$ *end-qj* (*s2*, *h2*) *Some h2'* = *Some hq2* $\oplus$ *Some hj2'* (*s2'*, *hj2'*), (*s2'*, *hj2'*) $\models$ *?J'*
hqg1 = *add-uguard-to-no-guard index hq1* (*?l s1'*) *hqg2* = *add-uguard-to-no-guard index hq2* (*?l s2'*)
using Σ'*-def* **by** *blast*
then obtain $\sigma1'$ $\sigma2'$ **where** $\sigma1' \in$ *start* (*s1*, *h1*) $\sigma2' \in$ *start* (*s2*, *h2*) (*s1'*,

h1') ∈ Σ *σ1'* (*s2'*, *h2'*) ∈ Σ *σ2'*
using *end-qj-def* **by** *blast*
then obtain *hj1 hj2* **where** *agrees* (− {*x*}) *s1* (*fst σ1'*) *Some* (*snd σ1'*) = *Some p1* ⊕ *Some hj1* (*fst σ1'*, *hj1*), (*fst σ1'*, *hj1*) ⊨ *?J*
agrees (− {*x*}) *s2* (*fst σ2'*) *Some* (*snd σ2'*) = *Some p2* ⊕ *Some hj2* (*fst σ2'*, *hj2*), (*fst σ2'*, *hj2*) ⊨ *?J*
using *start-def* ‹*remove-guards h1* = *p1*› ‹*remove-guards h2* = *p2*› **by** *force*

moreover have (*fst σ1'*, *hj1*), (*fst σ2'*, *hj2*) ⊨ *?J*
using *calculation*(*3*) *calculation*(*6*) *unaries unaryE* **by** *blast*
moreover have (*fst σ1'*, *p1*), (*fst σ2'*, *p2*) ⊨ *P*
proof −
have *fvA P* ⊆ − {*x*}
using *assms*(*5*) **by** *force*
then have *agrees* (*fvA P*) (*fst σ1'*) *s1* ∧ *agrees* (*fvA P*) (*fst σ2'*) *s2*
using *calculation*(*1*) *calculation*(*4*)
by (*metis agrees-comm agrees-union subset-Un-eq*)
then show *?thesis* **using** *r0*(*3*)
by (*meson agrees-same sat-comm*)
qed

ultimately have *σ1'*, *σ2'* ⊨ *Star P ?J* **using** *hyper-sat.simps*(*4*)[*of fst σ1' snd σ1' fst σ2' snd σ2'*] *prod.collapse*
by *metis*
then have *pair-sat* (Σ *σ1'*) (Σ *σ2'*) (*Star Q ?J'*)
using *asm0*(*2*)[*of σ1' σ2'*] **by** *blast*
then have (*s1'*, *h1'*), (*s2'*, *h2'*) ⊨ *Star Q ?J'*
using ‹(*s1'*, *h1'*) ∈ Σ *σ1'*› ‹(*s2'*, *h2'*) ∈ Σ *σ2'*› *pair-sat-def* **by** *blast*
moreover have (*s1'*, *hj1'*), (*s2'*, *hj2'*) ⊨ *?J'*
using *r*(*3*) *r*(*6*) *unaries unaryE* **by** *blast*
moreover have (*s1'*, *hq1*), (*s2'*, *hq2*) ⊨ *Q* **using** *magic-lemma*
using *calculation*(*1*) *calculation*(*2*) *precises r*(*2*) *r*(*5*) **by** *blast*
have (*s1'*, *add-uguard-to-no-guard index hq1* (*?l s1'*)), (*s2'*, *add-uguard-to-no-guard index hq2* (*?l s2'*)) ⊨ *Star Q ?G'*
proof (*rule no-guard-then-sat-star-uguard*)
show *no-guard hq1* ∧ *no-guard hq2*
using ‹(*s1'*, *hq1*), (*s2'*, *hq2*) ⊨ *Q*› *assms*(*11*) *no-guard-assertion-def* **by** *blast*
show (*s1'*, *hq1*), (*s2'*, *hq2*) ⊨ *Q*
using ‹(*s1'*, *hq1*), (*s2'*, *hq2*) ⊨ *Q*› **by** *auto*
qed
then show (*s1'*, *hqg1*), (*s2'*, *hqg2*) ⊨ *Star Q ?G'*
using *r*(*7*) *r*(*8*) **by** *force*
qed
then show *pair-sat* (*?Σ'* (*s1*, *h1*)) (*?Σ'* (*s2*, *h2*)) (*Star Q ?G'*)
proof (*rule pair-sat-close-var-double*)
show *x* ∉ *fvA* (*Star Q* (*UniqueGuard index* (λ*s*. *map-to-arg* (*s uarg*) # *map-to-list* (*s l*))))

using *assms*(*5*) *assms*(*9*) **by** *auto*
qed
qed

fix *pre-s h k*
assume (*pre-s*, *h*), (*pre-s*, *h*) ⊨ *Star P ?G*
then obtain *pp gg* **where** *Some h* = *Some pp* ⊕ *Some gg* (*pre-s*, *pp*), (*pre-s*, *pp*) ⊨ *P* (*pre-s*, *gg*), (*pre-s*, *gg*) ⊨ *?G*
using *always-sat-refl hyper-sat.simps*(*4*) **by** *blast*
then have *remove-guards h* = *pp*
using *assms*(*10*) *hyper-sat.simps*(*13*) *no-guard-and-no-heap no-guard-assertion-def* **by** *metis*
then have (*pre-s*, *remove-guards h*), (*pre-s*, *remove-guards h*) ⊨ *P*
using ‹(*pre-s*, *pp*), (*pre-s*, *pp*) ⊨ *P*› *hyper-sat.simps*(*9*) **by** *blast*
then have (*pre-s*, *remove-guards h*), (*pre-s*, *remove-guards h*) ⊨ *P*
by (*simp add*: *no-guard-remove-guards*)

show *safe k* (*Some* Γ) (*Catomic C*) (*pre-s*, *h*) (*?Σ′* (*pre-s*, *h*))
proof (*cases k*)
case (*Suc n*)
moreover have *safe* (*Suc n*) (*Some* Γ) (*Catomic C*) (*pre-s*, *h*) (*?Σ′* (*pre-s*, *h*))
proof (*rule safeSomeAltI*)
show *Catomic C* = *Cskip* ⟹ (*pre-s*, *h*) ∈ *?Σ′* (*pre-s*, *h*) **by** *simp*

fix *H hf hj v0*

assume *asm2*: *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *semi-consistent* Γ *v0 H* ∧ *sat-inv pre-s hj* Γ

define *v* **where** *v* = *f* (*normalize* (*get-fh H*))
define *s* **where** *s* = *pre-s*(*x* := *v*)
then have *v* = *s x* **by** *simp*
moreover have *agreements*: *agrees* (*fvC C* ∪ *fvA P* ∪ *fvA Q* ∪ *fvA J* ∪ *fvA* (*UniqueGuard k ?pre-l*)) *s pre-s*
by (*metis UnE agrees-comm agrees-update assms*(*5*) *assms*(*8*) *fvA.simps*(*9*) *s-def*)
have *asm1*: (*s*, *h*), (*s*, *h*) ⊨ *Star P ?G*
using *Un-iff*[*of x*] ‹(*pre-s*, *h*), (*pre-s*, *h*) ⊨ *Star P* (*UniqueGuard index* (λ*s*. *map-to-list* (*s l*)))›
agrees-same agrees-update[*of x*] *always-sat-refl assms*(*5*) *assms*(*8*) *fvA.simps*(*3*)[*of P UniqueGuard index* (λ*s*. *map-to-list* (*s l*))]
fvA.simps(*9*)[*of index* (λ*s*. *map-to-list* (*s l*))] *s-def*
by *metis*
moreover have *asm2-bis*: *sat-inv s hj* Γ
proof (*rule sat-inv-agrees*)
show *sat-inv pre-s hj* Γ **using** *asm2* **by** *simp*
show *agrees* (*fvA* (*invariant* Γ)) *pre-s s*
using *assms*(*1*) *assms*(*5*) *s-def*

by (*simp add*: *agrees-update*)
qed
moreover have (*s*, *remove-guards h*), (*s*, *remove-guards h*) $\models$ *P*
by (*meson* ‹(*pre-s*, *remove-guards h*), (*pre-s*, *remove-guards h*) $\models$ *P*› *agreements agrees-same agrees-union always-sat-refl*)

moreover have *agrees* ($-$ {*x*}) *pre-s s*
proof (*rule agreesI*)
fix *y* **assume** *y* $\in$ $-$ {*x*}
then have *y* $\neq$ *x*
by *force*
then show *pre-s y* = *s y*
by (*simp add*: *s-def*)
qed

moreover obtain (*s*, *pp*), (*s*, *pp*) $\models$ *P* (*s*, *gg*), (*s*, *gg*) $\models$ *?G*
by (*metis* ‹(*pre-s*, *gg*), (*pre-s*, *gg*) $\models$ *UniqueGuard index* (λ*s*. *map-to-list* (*s l*))› ‹*remove-guards h* = *pp*› *agrees-same-aux agrees-update always-sat-refl-aux assms*(*8*) *calculation*(*4*) *fvA*.*simps*(*9*) *s-def*)

let *?hf* = *remove-guards hf*
let *?H* = *remove-guards H*
let *?h* = *remove-guards h*

obtain *hhj* **where** *Some hhj* = *Some h* $\oplus$ *Some hj*
by (*metis asm2 plus*.*simps*(*2*) *plus*.*simps*(*3*) *plus-comm*)
then have *Some H* = *Some hhj* $\oplus$ *Some hf*
using *asm2* **by** *presburger*
then have *Some* (*remove-guards hhj*) = *Some ?h* $\oplus$ *Some hj*
by (*metis* ‹*Some hhj* = *Some h* $\oplus$ *Some hj*› *asm2 no-guards-remove remove-guards-sum sat-inv-def*)

moreover have *f* (*normalize* (*get-fh hj*)) = *v*
proof $-$
have *view* Γ (*normalize* (*get-fh hj*)) = *view* Γ (*normalize* (*get-fh H*))
using *assms*(*4*) *view-function-of-invE*
by (*metis* (*no-types*, *opaque-lifting*) ‹*Some hhj* = *Some h* $\oplus$ *Some hj*› *asm2 larger-def larger-trans plus-comm*)
then show *?thesis* **using** *assms*(*1*) *v-def* **by** *fastforce*
qed

then have (*s*, *hj*), (*s*, *hj*) $\models$ *?J*
by (*metis* ‹*v* = *s x*› *asm2-bis assms*(*1*) *hyper-sat*.*simps*(*11*) *sat-inv-def select-convs*(*5*))

ultimately have (*s*, *remove-guards hhj*), (*s*, *remove-guards hhj*) $\models$ *Star P ?J*
using ‹(*s*, *remove-guards h*), (*s*, *remove-guards h*) $\models$ *P*› *hyper-sat*.*simps*(*4*)

by *blast*
moreover have *bounded hhj*
apply (*rule bounded-smaller-sum*[*of H*])
apply (*metis asm2 full-ownership-then-bounded get-fh.simps*)
using ‹*Some H* = *Some hhj* ⊕ *Some hf*› **by** *blast*

ultimately have *all-safes*: $\bigwedge n$. *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C* (*s*, *remove-guards hhj*) (Σ (*s*, *remove-guards hhj*))
using *asm0*(*1*) **unfolding** *bounded-def remove-guards-def* **by** *simp*
then have $\bigwedge$*σ1 H1 σ2 H2 s2 C2*. *red-rtrans C σ1 C2 σ2* ⟹ *σ1* = (*s*, *H1*) ⟹ *σ2* = (*s2*, *H2*) ⟹
?H = *denormalize H1* ⟹
¬ *aborts C2 σ2* ∧ (*C2* = *Cskip* ⟶ (∃ *h1 H'*. *Some H'* = *Some h1* ⊕ *Some ?hf* ∧ *H2* = *FractionalHeap.normalize* (*get-fh H'*)
∧ *no-guard H'* ∧ *full-ownership* (*get-fh H'*) ∧ (*s2*, *h1*) ∈ Σ (*s*, *remove-guards hhj*)))
proof −
fix *σ1 H1 σ2 H2 s2 C2*
assume *?H* = *denormalize H1*
assume *red-rtrans C σ1 C2 σ2 σ1* = (*s*, *H1*) *σ2* = (*s2*, *H2*)

then show ¬ *aborts C2 σ2* ∧
(*C2* = *Cskip* ⟶
(∃ *h1 H'*.
Some H' = *Some h1* ⊕ *Some* (*remove-guards hf*) ∧
H2 = *FractionalHeap.normalize* (*get-fh H'*) ∧ *no-guard H'* ∧ *full-ownership* (*get-fh H'*) ∧ (*s2*, *h1*) ∈ Σ (*s*, *remove-guards hhj*)))
using *all-safes*
proof (*rule safe-atomic*)
show *?H* = *denormalize H1* **using** ‹*?H* = *denormalize H1*› **by** *simp*
show *Some ?H* = *Some* (*remove-guards hhj*) ⊕ *Some ?hf*
using ‹*Some H* = *Some hhj* ⊕ *Some hf*› *remove-guards-sum* **by** *blast*
show *full-ownership* (*get-fh* (*remove-guards H*)) ∧ *no-guard* (*remove-guards H*)
by (*metis asm2 get-fh-remove-guards no-guard-remove-guards*)
qed
qed
moreover have *?H* = *denormalize* (*normalize* (*get-fh H*))
by (*metis asm2 denormalize-properties*(*5*))
ultimately have *safe-atomic-simplified*: $\bigwedge$*σ2 H2 s2 C2*. *red-rtrans C* (*s*, *normalize* (*get-fh H*)) *C2 σ2*
⟹ *σ2* = (*s2*, *H2*) ⟹ ¬ *aborts C2 σ2* ∧ (*C2* = *Cskip* ⟶ (∃ *h1 H'*. *Some H'* = *Some h1* ⊕ *Some ?hf* ∧ *H2* = *FractionalHeap.normalize* (*get-fh H'*)
∧ *no-guard H'* ∧ *full-ownership* (*get-fh H'*) ∧ (*s2*, *h1*) ∈ Σ (*s*, *remove-guards hhj*)))
by *presburger*

have ¬ *aborts* (*Catomic C*) (*s*, *normalize* (*get-fh H*))
proof (*rule ccontr*)

assume ¬ ¬ *aborts* (*Catomic C*) (*s*, *normalize* (*get-fh H*))
then obtain *C′ σ′* **where** *asm3*: *red-rtrans C* (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′ σ′*
aborts C′ σ′
using *abort-atomic-cases* **by** *blast*
then have ¬ *aborts C′ σ′* **using** *safe-atomic-simplified*[*of C′ σ′ fst σ′ snd σ′*] **by** *simp*
then show *False* **using** *asm3*(*2*) **by** *simp*
qed
then show ¬ *aborts* (*Catomic C*) (*pre-s*, *normalize* (*get-fh H*))
by (*metis agreements aborts-agrees agrees-comm agrees-union fst-eqD fvC.simps*(*11*) *snd-conv*)

fix *C′ pre-s′ h′*
assume *red* (*Catomic C*) (*pre-s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*pre-s′*, *h′*)
then obtain *s′* **where** *red* (*Catomic C*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)
agrees (− {*x*}) *s′ pre-s′*
by (*metis* (*no-types*, *lifting*) *UnI1* ‹*agrees* (− {*x*}) *pre-s s*› *agrees-comm assms*(*5*) *fst-eqD fvC.simps*(*11*) *red-agrees snd-conv subset-Compl-singleton*)

then obtain *h1 H′* **where** *asm3*: *Some H′* = *Some h1* ⊕ *Some* (*remove-guards hf*) *C′* = *Cskip*
h′ = *FractionalHeap.normalize* (*get-fh H′*) *no-guard H′* ∧ *full-ownership* (*get-fh H′*) (*s′*, *h1*) ∈ Σ (*s*, *remove-guards hhj*)
using *safe-atomic-simplified*[*of C′* (*s′*, *h′*) *s′ h′*] **by** (*metis red-atomic-cases*)

moreover have *s x* = *s′ x* ∧ *s′ uarg* = *s uarg* ∧ *s l* = *s′ l* **using** *red-not-in-fv-not-touched*
using ‹*red* (*Catomic C*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)›
by (*metis Un-iff assms*(*5*) *assms*(*6*) *assms*(*7*) *fst-conv fvC.simps*(*11*))
have ∃ *hq′ hj′*. *Some h1* = *Some hq′* ⊕ *Some hj′* ∧ (*s′*, *add-uguard-to-no-guard index hq′* (*?l s′*)) ∈ Σ′ (*pre-s*, *h*) ∧ *sat-inv s′ hj′* Γ
∧ *f* (*normalize* (*get-fh hj′*)) = *uact index* (*s′ x*) (*map-to-arg* (*s′ uarg*))
proof −

have *pair-sat* (Σ (*s*, *remove-guards hhj*)) (Σ (*s*, *remove-guards hhj*)) (*Star Q ?J′*)
using *asm0*(*2*)[*of* (*s*, *remove-guards hhj*) (*s*, *remove-guards hhj*)]
using ‹(*s*, *remove-guards hhj*), (*s*, *remove-guards hhj*) ⊨ *Star P ?J*› **by** *blast*
then have (*s′*, *h1*), (*s′*, *h1*) ⊨ *Star Q ?J′*
using *asm3*(*5*) *pair-sat-def* **by** *blast*
then obtain *hq′ hj′* **where** *Some h1* = *Some hq′* ⊕ *Some hj′* (*s′*, *hq′*), (*s′*, *hq′*) ⊨ *Q* (*s′*, *hj′*), (*s′*, *hj′*) ⊨ *?J′*
using *always-sat-refl hyper-sat.simps*(*4*) **by** *blast*
then have *no-guard hj′*

by (*metis* (*no-types*, *opaque-lifting*) *calculation*(*1*) *calculation*(*4*) *no-guard-then-smaller-same plus-comm*)
moreover have f (*normalize* (*get-fh hj'*)) = *uact index* (*s' x*) (*map-to-arg* (*s' uarg*))
using ‹(*s'*, *hj'*), (*s'*, *hj'*) $\models$ *View f J* (λ*s*. *uact index* (*s x*) (*map-to-arg* (*s uarg*)))› **by** *auto*
moreover have (*s*, *remove-guards hhj*) $\in$ *start* (*pre-s*, *h*)
proof −
have *Some* (*remove-guards hhj*) = *Some ?h* $\oplus$ *Some hj*
using ‹*Some* (*remove-guards hhj*) = *Some* (*remove-guards h*) $\oplus$ *Some hj*› **by** *blast*
moreover have (*s*, *hj*), (*s*, *hj*) $\models$ *?J*
using ‹(*s*, *hj*), (*s*, *hj*) $\models$ *?J*› **by** *fastforce*
ultimately show *?thesis* **using** *start-def*
using ‹*agrees* (− {*x*}) *pre-s s*› **by** *fastforce*
qed
then have (*s'*, *h1*) $\in$ *end-qj* (*pre-s*, *h*)
using ‹*end-qj* $\equiv \lambda\sigma. \bigcup$ (Σ ‘ *start* σ)› *asm3*(*5*) **by** *blast*

then have (*s'*, *add-uguard-to-no-guard index hq'* (*?l s'*)) $\in \Sigma'$ (*pre-s*, *h*)
using Σ'*-def* ‹(*s'*, *hj'*), (*s'*, *hj'*) $\models$ *?J'*› ‹*Some h1* = *Some hq'* $\oplus$ *Some hj'*› **by** *blast*
ultimately show $\exists$ *hq' hj'*.
Some h1 = *Some hq'* $\oplus$ *Some hj'* $\wedge$
(*s'*, *add-uguard-to-no-guard index hq'* (*map-to-arg* (*s' uarg*) # *map-to-list* (*s' l*))) $\in \Sigma'$ (*pre-s*, *h*) $\wedge$
sat-inv s' hj' Γ $\wedge$ f (*FractionalHeap.normalize* (*get-fh hj'*)) = *uact index* (*s' x*) (*map-to-arg* (*s' uarg*))
using ‹(*s'*, *hj'*), (*s'*, *hj'*) $\models$ *?J'*› ‹*Some h1* = *Some hq'* $\oplus$ *Some hj'*›
assms(*1*) *hyper-sat.simps*(*11*) *sat-inv-def select-convs*(*5*)
by *fastforce*
qed
then obtain *hq' hj'* **where** *Some h1* = *Some hq'* $\oplus$ *Some hj'* (*s'*, *add-uguard-to-no-guard index hq'* (*?l s'*)) $\in \Sigma'$ (*pre-s*, *h*) *sat-inv s' hj'* Γ
f (*normalize* (*get-fh hj'*)) = *uact index* (*s' x*) (*map-to-arg* (*s' uarg*))
by *blast*
then have *safe n* (*Some* Γ) *C'* (*s'*, *add-uguard-to-no-guard index hq'* (*?l s'*)) (Σ' (*pre-s*, *h*))
using *asm3*(*2*) *safe-skip* **by** *blast*

moreover have $\exists$ *H''*. *semi-consistent* Γ *v0 H''* $\wedge$ *Some H''* = *Some* (*add-uguard-to-no-guard index hq'* (*?l s'*)) $\oplus$ *Some hj'* $\oplus$ *Some hf*
proof −
have *Some* (*add-uguard-to-no-guard index hq'* (*?l s'*)) = *Some hq'* $\oplus$ *Some* (*Map.empty*, *None*, [*index* $\mapsto$ *?l s'*])
by (*metis* ‹*Some h1* = *Some hq'* $\oplus$ *Some hj'*› *add-uguard-as-sum calculation*(*1*) *calculation*(*4*) *no-guard-then-smaller-same*)

obtain *hhf* **where** *Some hhf* = *Some h* $\oplus$ *Some hf*

by (*metis* (*no-types*, *opaque-lifting*) ‹*Some H* = *Some hhj* ⊕ *Some hf*› ‹*Some hhj* = *Some h* ⊕ *Some hj*› *option.exhaust-sel plus.simps*(*1*) *plus-asso plus-comm*)

then have *all-guards hhf*

by (*metis* (*no-types*, *lifting*) *all-guards-no-guard-propagates asm2 plus-asso plus-comm sat-inv-def semi-consistent-def*)

moreover have *get-gs h* = *None* ∧ *get-gu h index* = *Some* (*?pre-l s*)

proof −

have *no-guard pp*

using ‹*remove-guards h* = *pp*› *no-guard-remove-guards* **by** *blast*

then show *?thesis*

by (*metis* (*no-types*, *lifting*) ‹*Some h* = *Some pp* ⊕ *Some gg*› ‹$\bigwedge$*thesis*. (⟦(*s*, *pp*), (*s*, *pp*) ⊨ *P*; (*s*, *gg*), (*s*, *gg*) ⊨ *UniqueGuard index* (λ*s*. *map-to-list* (*s l*))⟧ ⟹ *thesis*) ⟹ *thesis*› *full-uguard-sum-same hyper-sat.simps*(*13*) *no-guard-remove*(*1*) *plus-comm*)

qed

moreover have $\bigwedge$*i'*. *i'* ≠ *index* ⟹ *get-gu h i'* = *None*

by (*metis* ‹*Some h* = *Some pp* ⊕ *Some gg*› ‹$\bigwedge$*thesis*. (⟦(*s*, *pp*), (*s*, *pp*) ⊨ *P*; (*s*, *gg*), (*s*, *gg*) ⊨ *UniqueGuard index* (λ*s*. *map-to-list* (*s l*))⟧ ⟹ *thesis*) ⟹ *thesis*› ‹*remove-guards h* = *pp*› *hyper-sat.simps*(*13*) *no-guard-remove*(*2*) *no-guard-remove-guards plus-comm*)

then obtain *sargs* **where** *get-gu hf index* = *None* ∧ *get-gs hf* = *Some* (*pwrite*, *sargs*)

by (*metis* (*no-types*, *opaque-lifting*) ‹*Some hhf* = *Some h* ⊕ *Some hf*› *add-gs.simps*(*1*) *all-guards-def calculation*(*1*) *calculation*(*2*) *compatible-def compatible-eq option.distinct*(*1*) *plus-extract*(*2*))

moreover obtain *uargs* **where** $\bigwedge$*i'*. *i'* ≠ *index* ⟹ *get-gu hf i'* = *Some* (*uargs i'*)

by (*metis* (*no-types*, *opaque-lifting*) ‹*Some hhf* = *Some h* ⊕ *Some hf*› ‹$\bigwedge$*i'*. *i'* ≠ *index* ⟹ *get-gu h i'* = *None*› *add-gu-def add-gu-single.simps*(*1*) *all-guards-exists-uargs calculation*(*1*) *plus-extract*(*3*))

then obtain *ghf* **where** *ghf-def*: *Some hf* = *Some* (*remove-guards hf*) ⊕ *Some ghf*

get-fh ghf = *Map.empty get-gu ghf index* = *None*

get-gs ghf = *Some* (*pwrite*, *sargs*) $\bigwedge$*i'*. *i'* ≠ *index* ⟹ *get-gu ghf i'* = *Some* (*uargs i'*)

using *decompose-guard-remove-easy*[*of hf*]

using *calculation*(*3*) **by** *auto*

have (*Map.empty*, *None*, [*index* ↦ *?l s'*]) ## *ghf*

proof (*rule compatibleI*)

show *compatible-fract-heaps* (*get-fh* (*Map.empty*, *None*, [*index* ↦ *map-to-arg* (*s' uarg*) # *map-to-list* (*s' l*)])) (*get-fh ghf*)

using *compatible-fract-heapsI* **by** *fastforce*

show $\bigwedge$*k*. *get-gu* (*Map.empty*, *None*, [*index* ↦ *map-to-arg* (*s' uarg*) # *map-to-list* (*s' l*)]) *k* = *None* ∨ *get-gu ghf k* = *None*

using *ghf-def*(*3*) **by** *auto*

qed (*simp*)

then obtain *g* **where** *g-def*: *Some g* = *Some* (*Map.empty*, *None*, [*index* ↦ *?l s′*]) ⊕ *Some ghf*
by *simp*
moreover have *H′* ## *g*
proof (*rule compatibleI*)
have *get-fh g* = *add-fh Map.empty Map.empty*
using *add-get-fh*[*of g* (*Map.empty*, *None*, [*index* ↦ *?l s′*]) *ghf*]
g-def ‹*get-fh ghf* = *Map.empty*›
by *fastforce*
then have *get-fh g* = *Map.empty*
using *add-fh-map-empty* **by** *auto*
then show *compatible-fract-heaps* (*get-fh H′*) (*get-fh g*)
using *compatible-fract-heapsI* **by** *force*
show ⋀*k*. *get-gu H′ k* = *None* ∨ *get-gu g k* = *None*
by (*meson asm3*(*4*) *no-guard-def*)
show ⋀*p p′*. *get-gs H′* = *Some p* ∧ *get-gs g* = *Some p′* ⟹ *pgte pwrite* (*padd* (*fst p*) (*fst p′*))
by (*metis asm3*(*4*) *no-guard-def option.simps*(*3*))
qed
then obtain *H″* **where** *Some H″* = *Some H′* ⊕ *Some g*
by *simp*
then have *Some H″* = *Some* (*add-uguard-to-no-guard index hq′* (*?l s′*)) ⊕ *Some hj′* ⊕ *Some hf*
proof −
have *Some H″* = *Some h1* ⊕ *Some g* ⊕ *Some* (*remove-guards hf*)
by (*metis* ‹*Some H″* = *Some H′* ⊕ *Some g*› *asm3*(*1*) *plus-comm simpler-asso*)
moreover have *Some* (*add-uguard-to-no-guard index hq′* (*?l s′*)) = *Some hq′* ⊕ *Some* (*Map.empty*, *None*, [*index* ↦ *?l s′*])
using ‹*Some* (*add-uguard-to-no-guard index hq′* (*map-to-arg* (*s′ uarg*) # *map-to-list* (*s′ l*))) = *Some hq′* ⊕ *Some* (*Map.empty*, *None*, [*index* ↦ *map-to-arg* (*s′ uarg*) # *map-to-list* (*s′ l*)])› **by** *blast*
ultimately show *?thesis*
by (*metis* (*no-types*, *lifting*) ‹*Some h1* = *Some hq′* ⊕ *Some hj′*› *g-def ghf-def*(*1*) *plus-comm simpler-asso*)
qed

moreover have *semi-consistent* Γ *v0 H″*
proof (*rule semi-consistentI*)
have *get-gs g* = *Some* (*pwrite*, *sargs*)
by (*metis full-sguard-sum-same g-def ghf-def*(*4*) *plus-comm*)
moreover have *get-gu g index* = *Some* (*?l s′*)
proof (*rule full-uguard-sum-same*)
show *get-gu* (*Map.empty*, *None*, [*index* ↦ *?l s′*]) *index* = *Some* (*?l s′*)
using *get-gu.simps* **by** *auto*
show *Some g* = *Some* (*Map.empty*, *None*, [*index* ↦ *?l s′*]) ⊕ *Some ghf*
using *g-def* **by** *auto*
qed
moreover have ⋀*i′*. *i′* ≠ *index* ⟹ *get-gu g i′* = *Some* (*uargs i′*)

by (*metis full-uguard-sum-same g-def ghf-def*(*5*) *plus-comm*)
ultimately have *all-guards g*
by (*metis all-guardsI option.discI*)
then show *all-guards* H''
by (*metis* ‹*Some* H'' = *Some* H' ⊕ *Some g*› *all-guards-same plus-comm*)
show *reachable* Γ *v0* H''
proof (*rule reachableI*)
fix *sargs*′ *uargs*′
assume *get-gs* H'' = *Some* (*pwrite*, *sargs*′) ∧ (∀ *k*. *get-gu* H'' *k* = *Some* (*uargs*′ *k*))
then have *sargs* = *sargs*′
by (*metis* (*no-types*, *opaque-lifting*) *Pair-inject* ‹*Some* H'' = *Some* H' ⊕ *Some g*› ‹*get-gs g* = *Some* (*pwrite*, *sargs*)› *full-sguard-sum-same option.inject plus-comm*)
moreover have *uargs*′ *index* = *?l s*′
by (*metis* ‹*Some* H'' = *Some* H' ⊕ *Some g*› ‹*get-gs* H'' = *Some* (*pwrite*, *sargs*′) ∧ (∀ *k*. *get-gu* H'' *k* = *Some* (*uargs*′ *k*))› ‹*get-gu g index* = *Some* (*map-to-arg* (*s*′ *uarg*) # *map-to-list* (*s*′ *l*))› *asm3*(*4*) *no-guard-remove*(*2*) *option.inject plus-comm*)
moreover have ⋀*i*′. *i*′ ≠ *index* ⟹ *uargs*′ *i*′ = *uargs i*′
by (*metis* ‹*Some* H'' = *Some* H' ⊕ *Some g*› ‹⋀*i*′. *i*′ ≠ *index* ⟹ *get-gu g i*′ = *Some* (*uargs i*′)› ‹*get-gs* H'' = *Some* (*pwrite*, *sargs*′) ∧ (∀ *k*. *get-gu* H'' *k* = *Some* (*uargs*′ *k*))› *asm3*(*4*) *no-guard-remove*(*2*) *option.sel plus-comm*)
moreover have *view* Γ (*FractionalHeap.normalize* (*get-fh hj*′)) = *view* Γ (*FractionalHeap.normalize* (*get-fh* H''))
using *assms*(*4*) ‹*sat-inv s*′ *hj*′ Γ›
proof (*rule view-function-of-invE*)
show $H'' \succeq hj'$
by (*metis* (*no-types*, *opaque-lifting*) ‹*Some* H'' = *Some* H' ⊕ *Some g*› ‹*Some h1* = *Some hq*′ ⊕ *Some hj*′› *asm3*(*1*) *larger-def larger-trans plus-comm*)
qed
moreover have *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs* (*uargs*(*index* := *?l s*′)) (*uact index* (*s*′ *x*) (*map-to-arg* (*s*′ *uarg*)))
proof −

have *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs* (*uargs*(*index* := *?pre-l s*′)) (*view* Γ (*FractionalHeap.normalize* (*get-fh H*)))
proof −
have *reachable* Γ *v0 H*
by (*meson asm2 semi-consistent-def*)
moreover have *get-gs H* = *Some* (*pwrite*, *sargs*)
by (*metis* ‹*Some H* = *Some hhj* ⊕ *Some hf*› ‹*get-gu hf index* = *None* ∧ *get-gs hf* = *Some* (*pwrite*, *sargs*)› *full-sguard-sum-same plus-comm*)
moreover have *get-gu H index* = *Some* (*?pre-l s*′)
by (*metis* ‹*Some H* = *Some hhj* ⊕ *Some hf*› ‹*Some hhj* = *Some h* ⊕ *Some hj*› ‹*get-gs h* = *None* ∧ *get-gu h index* = *Some* (*map-to-list* (*s l*))› ‹*s x* = *s*′ *x* ∧ *s*′ *uarg* = *s uarg* ∧ *s l* = *s*′ *l*› *full-uguard-sum-same*)
moreover have ⋀*i*. *i* ≠ *index* ⟹ *get-gu H i* = *Some* (*uargs i*)
by (*metis* ‹*Some H* = *Some hhj* ⊕ *Some hf*› ‹⋀*i*′. *i*′ ≠ *index* ⟹

get-gu hf i′ = Some (uargs i′)› full-uguard-sum-same plus-comm)
ultimately show *?thesis*
by (*simp add*: *reachable-def*)
qed
moreover have *view* Γ (*FractionalHeap.normalize* (*get-fh hj*)) = *view* Γ (*FractionalHeap.normalize* (*get-fh H*))
using *assms*(*4*)
proof (*rule view-function-of-invE*)
show *sat-inv s hj* Γ
by (*simp add*: *asm2-bis*)
show $H \succeq hj$
by (*metis* (*no-types*, *opaque-lifting*) ‹*Some H* = *Some hhj* ⊕ *Some hf*› ‹*Some hhj* = *Some h* ⊕ *Some hj*› *larger-def larger-trans plus-comm*)
qed
moreover have *s′ x* = *v*
using ‹*s x* = *s′ x* ∧ *s′ uarg* = *s uarg* ∧ *s l* = *s′ l*› ‹*v* = *s x*› **by** *presburger*
ultimately have *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs* (*uargs*(*index* := *?pre-l s′*)) *v*
using ‹*f* (*FractionalHeap.normalize* (*get-fh hj*)) = *v*› *assms*(*1*) **by** *auto*
then show *?thesis*
by (*metis UniqueStep* ‹*s′ x* = *v*› *assms*(*1*) *fun-upd-same fun-upd-upd select-convs*(*4*))
qed
moreover have *uargs′* = (*uargs*(*index* := *map-to-arg* (*s′ uarg*) # *map-to-list* (*s′ l*)))
proof (*rule ext*)
fix *i* **show** *uargs′ i* = (*uargs*(*index* := *map-to-arg* (*s′ uarg*) # *map-to-list* (*s′ l*))) *i*
apply (*cases i* = *index*)
using *calculation*(*2*) **apply** *auto*[*1*]
using *calculation*(*3*) **by** *force*
qed
ultimately show *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs′ uargs′* (*view* Γ (*FractionalHeap.normalize* (*get-fh H″*)))
using ‹*f* (*FractionalHeap.normalize* (*get-fh hj′*)) = *uact index* (*s′ x*) (*map-to-arg* (*s′ uarg*))› *assms*(*1*) **by** *force*
qed
qed
ultimately show ∃ *H″*. *semi-consistent* Γ *v0 H″* ∧ *Some H″* = *Some* (*add-uguard-to-no-guard index hq′* (*map-to-arg* (*s′ uarg*) # *map-to-list* (*s′ l*))) ⊕ *Some hj′* ⊕ *Some hf*
by *blast*
qed
ultimately obtain *H″* **where** *semi-consistent* Γ *v0 H″* ∧
Some H″ = *Some* (*add-uguard-to-no-guard index hq′* (*map-to-arg* (*s′ uarg*) # *map-to-list* (*s′ l*))) ⊕ *Some hj′* ⊕ *Some hf* **by** *blast*
moreover have *full-ownership* (*get-fh H″*) ∧ *h′* = *FractionalHeap.normalize*

$(get\text{-}fh\ H'')$

proof −

obtain x **where** $Some\ x = Some\ (add\text{-}uguard\text{-}to\text{-}no\text{-}guard\ index\ hq'\ (?l\ s')) \oplus Some\ hj'$

by $(metis\ calculation\ not\text{-}Some\text{-}eq\ plus.simps(1))$

then have $get\text{-}fh\ H'' = add\text{-}fh\ (add\text{-}fh\ (get\text{-}fh\ (add\text{-}uguard\text{-}to\text{-}no\text{-}guard\ index\ hq'\ (?l\ s')))\ (get\text{-}fh\ hj'))\ (get\text{-}fh\ hf)$

by $(metis\ add\text{-}get\text{-}fh\ calculation)$

moreover have $get\text{-}fh\ (add\text{-}uguard\text{-}to\text{-}no\text{-}guard\ index\ hq'\ (?l\ s')) = get\text{-}fh\ hq' \wedge get\text{-}fh\ hf = get\text{-}fh\ (remove\text{-}guards\ hf)$

by $(metis\ get\text{-}fh\text{-}add\text{-}uguard\ get\text{-}fh\text{-}remove\text{-}guards)$

ultimately show $?thesis$

by $(metis\ \langle Some\ h1 = Some\ hq' \oplus Some\ hj' \rangle\ add\text{-}get\text{-}fh\ asm3(1)\ asm3(3)\ asm3(4))$

qed

moreover have $sat\text{-}inv\ pre\text{-}s'\ hj'\ \Gamma$

proof $(rule\ sat\text{-}inv\text{-}agrees)$

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

by $(simp\ add\colon \langle sat\text{-}inv\ s'\ hj'\ \Gamma \rangle)$

show $agrees\ (fvA\ (invariant\ \Gamma))\ s'\ pre\text{-}s'$

using $UnCI\ \langle agrees\ (-\ \{x\})\ s'\ pre\text{-}s' \rangle\ assms(1)\ assms(5)\ select\text{-}convs(5)\ subset\text{-}Compl\text{-}singleton$

by $(metis\ agrees\text{-}union\ sup.orderE)$

qed

moreover have $safe\ n\ (Some\ \Gamma)\ C'\ (pre\text{-}s',\ add\text{-}uguard\text{-}to\text{-}no\text{-}guard\ index\ hq'\ (?l\ s'))\ (?\Sigma'\ (pre\text{-}s,\ h))$

proof $(rule\ safe\text{-}free\text{-}vars\text{-}Some)$

show $safe\ n\ (Some\ \Gamma)\ C'\ (s',\ add\text{-}uguard\text{-}to\text{-}no\text{-}guard\ index\ hq'\ (?l\ s'))\ (?\Sigma'\ (pre\text{-}s,\ h))$

by $(meson\ \langle safe\ n\ (Some\ \Gamma)\ C'\ (s',\ add\text{-}uguard\text{-}to\text{-}no\text{-}guard\ index\ hq'\ (map\text{-}to\text{-}arg\ (s'\ uarg)\ \#\ map\text{-}to\text{-}list\ (s'\ l)))\ (\Sigma'\ (pre\text{-}s,\ h)) \rangle\ close\text{-}var\text{-}subset\ safe\text{-}larger\text{-}set)$

show $agrees\ (fvC\ C' \cup (-\ \{x\}) \cup fvA\ (invariant\ \Gamma))\ s'\ pre\text{-}s'$

by $(metis\ UnI2\ Un\text{-}absorb1\ \langle agrees\ (-\ \{x\})\ s'\ pre\text{-}s' \rangle\ asm3(2)\ assms(1)\ assms(5)\ empty\text{-}iff\ fvC.simps(1)\ inf\text{-}sup\text{-}aci(5)\ select\text{-}convs(5)\ subset\text{-}Compl\text{-}singleton)$

show $upper\text{-}fvs\ (close\text{-}var\ (\Sigma'\ (pre\text{-}s,\ h))\ x)\ (-\ \{x\})$

by $(simp\ add\colon upper\text{-}fvs\text{-}close\text{-}vars)$

qed

ultimately show $\exists\, h''\ H'\ hj'.$

$full\text{-}ownership\ (get\text{-}fh\ H') \wedge$

$semi\text{-}consistent\ \Gamma\ v0\ H' \wedge$

$sat\text{-}inv\ pre\text{-}s'\ hj'\ \Gamma \wedge h' = FractionalHeap.normalize\ (get\text{-}fh\ H') \wedge Some\ H' = Some\ h'' \oplus Some\ hj' \oplus Some\ hf$

$\wedge\ safe\ n\ (Some\ \Gamma)\ C'\ (pre\text{-}s',\ h'')\ (?\Sigma'\ (pre\text{-}s,\ h))$ **using** $\langle sat\text{-}inv\ s'\ hj'\ \Gamma \rangle$ **by** $blast$

qed $(simp)$

ultimately show $safe\ k\ (Some\ \Gamma)\ (Catomic\ C)\ (pre\text{-}s,\ h)\ (?\Sigma'\ (pre\text{-}s,\ h))$ **by** $blast$

qed $(simp)$

qed
qed

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

fixes *map-to-multiset* :: *nat* $\Rightarrow$ *'a multiset*
fixes *map-to-arg* :: *nat* $\Rightarrow$ *'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*. *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* (λ*s*. *map-to-multiset* (*s ms*))
and *x* $\notin$ *fvS* (λ*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 (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*))))
proof −

let *?J* = *View f J* (λ*s*. *s x*)
let *?J'* = *View f J* (λ*s*. *sact* (*s x*) (*map-to-arg* (*s sarg*)))
let *?pre-ms* = λ*s*. *map-to-multiset* (*s ms*)
let *?G* = *SharedGuard* π *?pre-ms*
let *?ms* = λ*s*. {# *map-to-arg* (*s sarg*) #} + *map-to-multiset* (*s ms*)
let *?G'* = *SharedGuard* π *?ms*

have *unaries*: *unary ?J* $\wedge$ *unary ?J'*
by (*simp add*: *assms*(*3*) *unary-inv-then-view*)
moreover have *precises*: *precise ?J* $\wedge$ *precise ?J'*
by (*simp add*: *assms*(*3*) *precise-inv-then-view*)

obtain Σ **where** *asm0*: $\bigwedge n\ \sigma.\ \sigma, \sigma \models$ *Star P ?J* $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C* σ ($\Sigma\ \sigma$)
$\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models$ *Star P ?J* $\Longrightarrow$ *pair-sat* ($\Sigma\ \sigma$) ($\Sigma\ \sigma'$) (*Star Q ?J'*)

using *assms*(2) *hoare-triple-valid-def* **by** *blast*

define *start* **where** $start = (\lambda\sigma.\ \{\ (s,\ h)\ |s\ h\ hj.\ agrees\ (-\ \{x\})\ (fst\ \sigma)\ s \wedge Some\ h = Some\ (remove\text{-}guards\ (snd\ \sigma)) \oplus Some\ hj \wedge (s,\ hj),\ (s,\ hj) \models\ ?J\})$
define *end-qj* **where** $end\text{-}qj = (\lambda\sigma.\ \bigcup\sigma' \in start\ \sigma.\ \Sigma\ \sigma')$
define Σ' **where** $\Sigma' = (\lambda\sigma.\ \{\ (s,\ add\text{-}sguard\text{-}to\text{-}no\text{-}guard\ hq\ \pi\ (?ms\ s))\ |s\ hq\ h\ hj.\ (s,\ h) \in end\text{-}qj\ \sigma \wedge Some\ h = Some\ hq \oplus Some\ hj \wedge (s,\ hj),\ (s,\ hj) \models\ ?J'\ \})$

let $?\Sigma' = \lambda\sigma.\ close\text{-}var\ (\Sigma'\ \sigma)\ x$

show *hoare-triple-valid* (*Some* Γ) (*Star P ?G*) (*Catomic C*) (*Star Q ?G*$'$)
proof (*rule hoare-triple-validI*)
show $\bigwedge s\ h\ s'\ h'.\ (s,\ h),\ (s',\ h') \models Star\ P\ ?G \Longrightarrow pair\text{-}sat\ (?\Sigma'\ (s,\ h))\ (?\Sigma'\ (s',\ h'))\ (Star\ Q\ ?G')$
proof −
fix *s1 h1 s2 h2*
assume *asm1*: $(s1,\ h1),\ (s2,\ h2) \models Star\ P\ ?G$
then obtain *p1 p2 g1 g2* **where** *r0*: $Some\ h1 = Some\ p1 \oplus Some\ g1$
$Some\ h2 = Some\ p2 \oplus Some\ g2$
$(s1,\ p1),\ (s2,\ p2) \models P\ (s1,\ g1),\ (s2,\ g2) \models\ ?G$
using *hyper-sat.simps*(4) **by** *auto*
then obtain $remove\text{-}guards\ h1 = p1\ remove\text{-}guards\ h2 = p2$
using *assms*(10) *hyper-sat.simps*(12) *no-guard-and-no-heap no-guard-assertion-def*
by *metis*

have $pair\text{-}sat\ (\Sigma'\ (s1,\ h1))\ (\Sigma'\ (s2,\ h2))\ (Star\ Q\ ?G')$
proof (*rule pair-satI*)
fix $s1'\ hqg1\ s2'\ hqg2\ \sigma2'$
assume *asm2*: $(s1',\ hqg1) \in \Sigma'\ (s1,\ h1) \wedge (s2',\ hqg2) \in \Sigma'\ (s2,\ h2)$
then obtain $h1'\ hj1'\ h2'\ hj2'\ hq1\ hq2$ **where** *r*: $(s1',\ h1') \in end\text{-}qj\ (s1,\ h1)\ Some\ h1' = Some\ hq1 \oplus Some\ hj1'$
$(s1',\ hj1'),\ (s1',\ hj1') \models\ ?J'\ (s2',\ h2') \in end\text{-}qj\ (s2,\ h2)\ Some\ h2' = Some\ hq2 \oplus Some\ hj2'\ (s2',\ hj2'),\ (s2',\ hj2') \models\ ?J'$
$hqg1 = add\text{-}sguard\text{-}to\text{-}no\text{-}guard\ hq1\ \pi\ (?ms\ s1')\ hqg2 = add\text{-}sguard\text{-}to\text{-}no\text{-}guard\ hq2\ \pi\ (?ms\ s2')$
using Σ'-*def* **by** *blast*
then obtain $\sigma1'\ \sigma2'$ **where** $\sigma1' \in start\ (s1,\ h1)\ \sigma2' \in start\ (s2,\ h2)\ (s1',\ h1') \in \Sigma\ \sigma1'\ (s2',\ h2') \in \Sigma\ \sigma2'$
using *end-qj-def* **by** *blast*
then obtain *hj1 hj2* **where** $agrees\ (-\ \{x\})\ s1\ (fst\ \sigma1')\ Some\ (snd\ \sigma1') = Some\ p1 \oplus Some\ hj1\ (fst\ \sigma1',\ hj1),\ (fst\ \sigma1',\ hj1) \models\ ?J$
$agrees\ (-\ \{x\})\ s2\ (fst\ \sigma2')\ Some\ (snd\ \sigma2') = Some\ p2 \oplus Some\ hj2\ (fst\ \sigma2',\ hj2),\ (fst\ \sigma2',\ hj2) \models\ ?J$
using *start-def* ‹$remove\text{-}guards\ h1 = p1$› ‹$remove\text{-}guards\ h2 = p2$› **by** *force*

moreover have $(fst\ \sigma1',\ hj1),\ (fst\ \sigma2',\ hj2) \models\ ?J$
using *calculation*(3) *calculation*(6) *unaries unaryE* **by** *blast*
moreover have $(fst\ \sigma1',\ p1),\ (fst\ \sigma2',\ p2) \models P$

proof −
have $fvA\ P \subseteq -\ \{x\}$
using $assms(5)$ **by** *force*
then have $agrees\ (fvA\ P)\ (fst\ \sigma 1')\ s1 \wedge agrees\ (fvA\ P)\ (fst\ \sigma 2')\ s2$
using $calculation(1)\ calculation(4)$
by (*metis agrees-comm agrees-union subset-Un-eq*)
then show *?thesis* **using** $r0(3)$
by (*meson agrees-same sat-comm*)
qed

ultimately have $\sigma 1', \sigma 2' \models Star\ P\ ?J$ **using** $hyper\text{-}sat.simps(4)[of\ fst\ \sigma 1'\ snd\ \sigma 1'\ fst\ \sigma 2'\ snd\ \sigma 2']$ *prod.collapse*
by *metis*
then have $pair\text{-}sat\ (\Sigma\ \sigma 1')\ (\Sigma\ \sigma 2')\ (Star\ Q\ ?J')$
using $asm0(2)[of\ \sigma 1'\ \sigma 2']$ **by** *blast*
then have $(s1', h1'), (s2', h2') \models Star\ Q\ ?J'$
using $‹(s1', h1') \in \Sigma\ \sigma 1'›\ ‹(s2', h2') \in \Sigma\ \sigma 2'›$ *pair-sat-def* **by** *blast*
moreover have $(s1', hj1'), (s2', hj2') \models ?J'$
using $r(3)\ r(6)$ *unaries unaryE* **by** *blast*
moreover have $(s1', hq1), (s2', hq2) \models Q$ **using** *magic-lemma*
using $calculation(1)\ calculation(2)\ precises\ r(2)\ r(5)$ **by** *blast*
moreover have $no\text{-}guard\ hq1 \wedge no\text{-}guard\ hq2$
using $assms(11)\ calculation(3)$ *no-guard-assertion-def* **by** *blast*
ultimately show $(s1', hqg1), (s2', hqg2) \models Star\ Q\ ?G'$
using *no-guard-then-sat-star* $r(7)\ r(8)$
by (*metis* (*mono-tags*, *lifting*))
qed
then show $pair\text{-}sat\ (?\Sigma'\ (s1, h1))\ (?\Sigma'\ (s2, h2))\ (Star\ Q\ ?G')$
proof (*rule pair-sat-close-var-double*)
show $x \notin fvA\ (Star\ Q\ (SharedGuard\ \pi\ (\lambda s.\ \{\#map\text{-}to\text{-}arg\ (s\ sarg)\#\} + map\text{-}to\text{-}multiset\ (s\ ms))))$
using $assms(5)\ assms(9)$ **by** *auto*
qed
qed

fix *pre-s h k*
assume $(pre\text{-}s, h), (pre\text{-}s, h) \models Star\ P\ ?G$
then obtain *pp gg* **where** $Some\ h = Some\ pp \oplus Some\ gg\ (pre\text{-}s, pp), (pre\text{-}s, pp) \models P\ (pre\text{-}s, gg), (pre\text{-}s, gg) \models ?G$
using *always-sat-refl* $hyper\text{-}sat.simps(4)$ **by** *blast*
then have $remove\text{-}guards\ h = pp$
by (*meson* $assms(10)\ hyper\text{-}sat.simps(12)$ *no-guard-and-no-heap no-guard-assertion-def*)
then have $(pre\text{-}s, remove\text{-}guards\ h), (pre\text{-}s, remove\text{-}guards\ h) \models P$
using $‹(pre\text{-}s, pp), (pre\text{-}s, pp) \models P›\ hyper\text{-}sat.simps(9)$ **by** *blast*
then have $(pre\text{-}s, remove\text{-}guards\ h), (pre\text{-}s, remove\text{-}guards\ h) \models P$
by (*simp add*: *no-guard-remove-guards*)

show $safe\ k\ (Some\ \Gamma)\ (Catomic\ C)\ (pre\text{-}s, h)\ (?\Sigma'\ (pre\text{-}s, h))$
proof (*cases k*)

case (*Suc n*)
moreover have *safe* (*Suc n*) (*Some* Γ) (*Catomic C*) (*pre-s*, *h*) (*?Σ′* (*pre-s*, *h*))
proof (*rule safeSomeAltI*)
show *Catomic C* = *Cskip* ⟹ (*pre-s*, *h*) ∈ *?Σ′* (*pre-s*, *h*) **by** *simp*

fix *H hf hj v0*

assume *asm2*: *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧ *semi-consistent* Γ *v0 H* ∧ *sat-inv pre-s hj* Γ

define *v* **where** *v* = *f* (*normalize* (*get-fh H*))
define *s* **where** *s* = *pre-s*(*x* := *v*)
then have *v* = *s x* **by** *simp*
moreover have *agreements*: *agrees* (*fvC C* ∪ *fvA P* ∪ *fvA Q* ∪ *fvA J* ∪ *fvA* (*SharedGuard* π (λ*s*. *map-to-multiset* (*s ms*)))) *s pre-s*
by (*metis* (*mono-tags*, *lifting*) *Un-iff agrees-def assms*(*5*) *assms*(*8*) *fun-upd-other fvA.simps*(*8*) *s-def*)
then have *asm1*: (*s*, *h*), (*s*, *h*) ⊨ *Star P ?G*
— 10s
by (*metis* (*mono-tags*, *lifting*) ‹(*pre-s*, *h*), (*pre-s*, *h*) ⊨ *Star P* (*SharedGuard* π (λ*s*. *map-to-multiset* (*s ms*)))› *agrees-same agrees-union fvA.simps*(*3*) *fvA.simps*(*8*) *sat-comm*)
moreover have *asm2-bis*: *sat-inv s hj* Γ
proof (*rule sat-inv-agrees*)
show *sat-inv pre-s hj* Γ **using** *asm2* **by** *simp*
show *agrees* (*fvA* (*invariant* Γ)) *pre-s s*
using *assms*(*1*) *assms*(*5*) *s-def*
by (*simp add*: *agrees-update*)
qed
moreover have (*s*, *remove-guards h*), (*s*, *remove-guards h*) ⊨ *P*
by (*meson* ‹(*pre-s*, *remove-guards h*), (*pre-s*, *remove-guards h*) ⊨ *P*› *agreements agrees-same agrees-union always-sat-refl*)
then have (*s*, *remove-guards h*), (*s*, *remove-guards h*) ⊨ *P*
by (*simp add*: *no-guard-remove-guards*)

moreover have *agrees* (− {*x*}) *pre-s s*
proof (*rule agreesI*)
fix *y* **assume** *y* ∈ − {*x*}
then have *y* ≠ *x*
by *force*
then show *pre-s y* = *s y*
by (*simp add*: *s-def*)
qed

moreover obtain (*s*, *pp*), (*s*, *pp*) ⊨ *P* (*s*, *gg*), (*s*, *gg*) ⊨ *?G*
using ‹(*pre-s*, *gg*), (*pre-s*, *gg*) ⊨ *SharedGuard* π (λ*s*. *map-to-multiset* (*s ms*))› ‹*remove-guards h* = *pp*› *agreements agrees-same agrees-union always-sat-refl-aux calculation*(*4*) **by** *blast*

let *?hf* = *remove-guards hf*
let *?H* = *remove-guards H*
let *?h* = *remove-guards h*

obtain *hhj* **where** *Some hhj* = *Some h* ⊕ *Some hj*
by (*metis asm2 plus.simps*(*2*) *plus.simps*(*3*) *plus-comm*)
then have *Some H* = *Some hhj* ⊕ *Some hf*
using *asm2* **by** *presburger*
then have *Some* (*remove-guards hhj*) = *Some ?h* ⊕ *Some hj*
by (*metis* ‹*Some hhj* = *Some h* ⊕ *Some hj*› *asm2 no-guards-remove remove-guards-sum sat-inv-def*)

moreover have *f* (*normalize* (*get-fh hj*)) = *v*
proof −
have *view* Γ (*normalize* (*get-fh hj*)) = *view* Γ (*normalize* (*get-fh H*))
using *assms*(*4*) *view-function-of-invE*
by (*metis* (*no-types*, *opaque-lifting*) ‹*Some hhj* = *Some h* ⊕ *Some hj*› *asm2 larger-def larger-trans plus-comm*)
then show *?thesis* **using** *assms*(*1*) *v-def* **by** *fastforce*
qed

then have (*s*, *hj*), (*s*, *hj*) ⊨ *?J*
by (*metis* ‹*v* = *s x*› *asm2-bis assms*(*1*) *hyper-sat.simps*(*11*) *sat-inv-def select-convs*(*5*))

ultimately have (*s*, *remove-guards hhj*), (*s*, *remove-guards hhj*) ⊨ *Star P ?J*
using ‹(*s*, *remove-guards h*), (*s*, *remove-guards h*) ⊨ *P*› *hyper-sat.simps*(*4*) **by** *blast*
moreover have *bounded hhj*
apply (*rule bounded-smaller-sum*[*OF* - ‹*Some H* = *Some hhj* ⊕ *Some hf*›])
by (*metis asm2 full-ownership-then-bounded get-fh.simps*)

ultimately have *all-safes*: ⋀*n*. *safe n* (*None* :: (′*i*, ′*a*, *nat*) *cont*) *C* (*s*, *remove-guards hhj*) (Σ (*s*, *remove-guards hhj*))
using *asm0*(*1*) **unfolding** *bounded-def remove-guards-def* **by** *simp*
then have ⋀*σ1 H1 σ2 H2 s2 C2*. *red-rtrans C σ1 C2 σ2* ⟹ *σ1* = (*s*, *H1*) ⟹ *σ2* = (*s2*, *H2*) ⟹
?H = *denormalize H1* ⟹
¬ *aborts C2 σ2* ∧ (*C2* = *Cskip* ⟶ (∃ *h1 H*′. *Some H*′ = *Some h1* ⊕ *Some ?hf* ∧ *H2* = *FractionalHeap.normalize* (*get-fh H*′)
∧ *no-guard H*′ ∧ *full-ownership* (*get-fh H*′) ∧ (*s2*, *h1*) ∈ Σ (*s*, *remove-guards hhj*)))
proof −
fix *σ1 H1 σ2 H2 s2 C2*
assume *?H* = *denormalize H1*
assume *red-rtrans C σ1 C2 σ2 σ1* = (*s*, *H1*) *σ2* = (*s2*, *H2*)

then show ¬ *aborts C2 σ2* ∧
(*C2* = *Cskip* ⟶
(∃ *h1 H′*.
Some H′ = *Some h1* ⊕ *Some* (*remove-guards hf*) ∧
H2 = *FractionalHeap.normalize* (*get-fh H′*) ∧ *no-guard H′* ∧ *full-ownership* (*get-fh H′*) ∧ (*s2*, *h1*) ∈ Σ (*s*, *remove-guards hhj*)))
using *all-safes*
proof (*rule safe-atomic*)
show *?H* = *denormalize H1* **using** ‹*?H* = *denormalize H1*› **by** *simp*
show *Some ?H* = *Some* (*remove-guards hhj*) ⊕ *Some ?hf*
using ‹*Some H* = *Some hhj* ⊕ *Some hf*› *remove-guards-sum* **by** *blast*
show *full-ownership* (*get-fh* (*remove-guards H*)) ∧ *no-guard* (*remove-guards H*)
by (*metis asm2 get-fh-remove-guards no-guard-remove-guards*)
qed
qed
moreover have *?H* = *denormalize* (*normalize* (*get-fh H*))
by (*metis asm2 denormalize-properties*(*5*))
ultimately have *safe-atomic-simplified*: ⋀*σ2 H2 s2 C2*. *red-rtrans C* (*s*, *normalize* (*get-fh H*)) *C2 σ2*
⟹ *σ2* = (*s2*, *H2*) ⟹ ¬ *aborts C2 σ2* ∧ (*C2* = *Cskip* ⟶ (∃ *h1 H′*. *Some H′* = *Some h1* ⊕ *Some ?hf* ∧ *H2* = *FractionalHeap.normalize* (*get-fh H′*)
∧ *no-guard H′* ∧ *full-ownership* (*get-fh H′*) ∧ (*s2*, *h1*) ∈ Σ (*s*, *remove-guards hhj*)))
by *presburger*

have ¬ *aborts* (*Catomic C*) (*s*, *normalize* (*get-fh H*))
proof (*rule ccontr*)
assume ¬ ¬ *aborts* (*Catomic C*) (*s*, *normalize* (*get-fh H*))
then obtain *C′ σ′* **where** *asm3*: *red-rtrans C* (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′ σ′*
aborts C′ σ′
using *abort-atomic-cases* **by** *blast*
then have ¬ *aborts C′ σ′* **using** *safe-atomic-simplified*[*of C′ σ′ fst σ′ snd σ′*] **by** *simp*
then show *False* **using** *asm3*(*2*) **by** *simp*
qed
then show ¬ *aborts* (*Catomic C*) (*pre-s*, *normalize* (*get-fh H*))
by (*metis agreements aborts-agrees agrees-comm agrees-union fst-eqD fvC.simps*(*11*) *snd-conv*)

fix *C′ pre-s′ h′*
assume *red* (*Catomic C*) (*pre-s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*pre-s′*, *h′*)
then obtain *s′* **where** *red* (*Catomic C*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)
agrees (− {*x*}) *s′ pre-s′*
by (*metis* (*no-types*, *lifting*) *UnI1* ‹*agrees* (− {*x*}) *pre-s s*› *agrees-comm*

assms(5) fst-eqD fvC.simps(11) red-agrees snd-conv subset-Compl-singleton)

then obtain *h1 H′* **where** *asm3*: *Some H′ = Some h1* $\oplus$ *Some (remove-guards hf) C′ = Cskip*
h′ = FractionalHeap.normalize (get-fh H′) no-guard H′ $\wedge$ *full-ownership (get-fh H′) (s′, h1)* $\in \Sigma$ *(s, remove-guards hhj)*
using *safe-atomic-simplified[of C′ (s′, h′) s′ h′]* **by** *(metis red-atomic-cases)*

moreover have *s x = s′ x* $\wedge$ *s sarg = s′ sarg* $\wedge$ *s ms = s′ ms* **using** *red-not-in-fv-not-touched*
using ‹*red (Catomic C) (s, FractionalHeap.normalize (get-fh H)) C′ (s′, h′)*›
by *(metis UnI1 assms(5) assms(6) assms(7) fst-eqD fvC.simps(11))*

have $\exists$ *hq′ hj′. Some h1 = Some hq′* $\oplus$ *Some hj′* $\wedge$ *(s′, add-sguard-to-no-guard hq′* π *(?ms s′))* $\in \Sigma'$ *(pre-s, h)*
$\wedge$ *sat-inv s′ hj′* Γ $\wedge$ *f (normalize (get-fh hj′)) = sact v (map-to-arg (s′ sarg))*
proof −
have *pair-sat* $(\Sigma$ *(s, remove-guards hhj))* $(\Sigma$ *(s, remove-guards hhj)) (Star Q ?J′)*
using *asm0(2)[of (s, remove-guards hhj) (s, remove-guards hhj)]*
using ‹*(s, remove-guards hhj), (s, remove-guards hhj)* $\models$ *Star P ?J*› **by** *blast*
then have *(s′, h1), (s′, h1)* $\models$ *Star Q ?J′*
using *asm3(5) pair-sat-def* **by** *blast*
then obtain *hq′ hj′* **where** *Some h1 = Some hq′* $\oplus$ *Some hj′ (s′, hq′), (s′, hq′)* $\models$ *Q (s′, hj′), (s′, hj′)* $\models$ *?J′*
using *always-sat-refl hyper-sat.simps(4)* **by** *blast*
then have *no-guard hj′*
by *(metis (no-types, opaque-lifting) calculation(1) calculation(4) no-guard-then-smaller-same plus-comm)*
moreover have *f (normalize (get-fh hj′)) = sact v (map-to-arg (s′ sarg))*
using ‹*(s′, hj′), (s′, hj′)* $\models$ *View f J* $(\lambda s.$ *sact (s x) (map-to-arg (s sarg)))*› ‹*s x = s′ x* $\wedge$ *s sarg = s′ sarg* $\wedge$ *s ms = s′ ms*› ‹*v = s x*› **by** *fastforce*
moreover have *(s, remove-guards hhj)* $\in$ *start (pre-s, h)*
proof −
have *Some (remove-guards hhj) = Some ?h* $\oplus$ *Some hj*
using ‹*Some (remove-guards hhj) = Some (remove-guards h)* $\oplus$ *Some hj*› **by** *blast*
moreover have *(s, hj), (s, hj)* $\models$ *?J*
using ‹*(s, hj), (s, hj)* $\models$ *?J*› **by** *fastforce*
ultimately show *?thesis* **using** *start-def*
using ‹*agrees* $(- \{x\})$ *pre-s s*› **by** *fastforce*
qed
then have *(s′, h1)* $\in$ *end-qj (pre-s, h)*
using ‹*end-qj* $\equiv \lambda\sigma. \bigcup (\Sigma$ ‘ *start* $\sigma)$› *asm3(5)* **by** *blast*

then have *(s′, add-sguard-to-no-guard hq′* π *(?ms s′))* $\in \Sigma'$ *(pre-s, h)*
using Σ'*-def* ‹*(s′, hj′), (s′, hj′)* $\models$ *?J′*› ‹*Some h1 = Some hq′* $\oplus$ *Some*

hj'› **by** *blast*
ultimately show $\exists hq'\ hj'$.
Some h1 = *Some hq'* ⊕ *Some hj'* ∧
(s', *add-sguard-to-no-guard* hq' π ({#*map-to-arg* (s' *sarg*)#} + *map-to-multiset* (s' *ms*))) ∈ Σ′ (*pre-s*, *h*) ∧
sat-inv s' hj' Γ ∧ *f* (*FractionalHeap.normalize* (*get-fh* hj')) = *sact v* (*map-to-arg* (s' *sarg*))
using ‹(s', hj'), (s', hj') ⊨ *View f J* (λ*s*. *sact* (*s x*) (*map-to-arg* (*s sarg*)))› ‹*Some h1* = *Some hq'* ⊕ *Some hj'*› *assms*(*1*) *sat-inv-def* **by** *fastforce*
qed
then obtain hq' hj' **where** *Some h1* = *Some hq'* ⊕ *Some hj'* (s', *add-sguard-to-no-guard* hq' π (*?ms* s')) ∈ Σ′ (*pre-s*, *h*) *sat-inv* s' hj' Γ
f (*FractionalHeap.normalize* (*get-fh* hj')) = *sact v* (*map-to-arg* (s' *sarg*))
by *blast*
then have *safe n* (*Some* Γ) C' (s', *add-sguard-to-no-guard* hq' π (*?ms* s')) (Σ′ (*pre-s*, *h*))
using *asm3*(*2*) *safe-skip* **by** *blast*

moreover have $\exists H''$. *semi-consistent* Γ *v0* H'' ∧ *Some* H'' = *Some* (*add-sguard-to-no-guard* hq' π (*?ms* s')) ⊕ *Some hj'* ⊕ *Some hf*
proof −
have *Some* (*add-sguard-to-no-guard* hq' π (*?ms* s')) = *Some hq'* ⊕ *Some* (*Map.empty*, *Some* (π, *?ms* s'), (λ-. *None*))
using ‹*Some h1* = *Some hq'* ⊕ *Some hj'*› *add-sguard-as-sum asm3*(*1*) *asm3*(*4*) *no-guard-then-smaller-same* **by** *blast*

obtain *hhf* **where** *Some hhf* = *Some h* ⊕ *Some hf*
by (*metis* (*no-types*, *opaque-lifting*) ‹*Some H* = *Some hhj* ⊕ *Some hf*› ‹*Some hhj* = *Some h* ⊕ *Some hj*› *option.exhaust-sel plus.simps*(*1*) *plus-asso plus-comm*)
then have *all-guards hhf*
by (*metis* (*no-types*, *lifting*) *all-guards-no-guard-propagates asm2 plus-asso plus-comm sat-inv-def semi-consistent-def*)

moreover have *get-gu h* = (λ-. *None*) ∧ *get-gs h* = *Some* (π, *?pre-ms s*)
proof −
have *no-guard pp*
using ‹(*pre-s*, *pp*), (*pre-s*, *pp*) ⊨ *P*› *assms*(*10*) *no-guard-assertion-def* **by** *blast*
then show *?thesis*
by (*metis* ‹*Some h* = *Some pp* ⊕ *Some gg*› ‹⋀*thesis*. (⟦(*s*, *pp*), (*s*, *pp*) ⊨ *P*; (*s*, *gg*), (*s*, *gg*) ⊨ *SharedGuard* π (λ*s*. *map-to-multiset* (*s ms*))⟧ ⟹ *thesis*) ⟹ *thesis*› ‹*remove-guards h* = *pp*› *decompose-heap-triple fst-conv hyper-sat.simps*(*12*) *no-guard-remove*(*2*) *plus-comm remove-guards-def snd-conv sum-gs-one-none*)
qed
then have $\exists \pi'$ *msf uargs*. (∀ *k*. *get-gu hf k* = *Some* (*uargs k*)) ∧
(π = *pwrite* ∧ *get-gs hf* = *None* ∧ *msf* = {#} ∨ *pwrite* = *padd* π π' ∧ *get-gs hf* = *Some* (π', *msf*))

using *all-guards-sum-known-one*[*of hhf h hf* π]
using ‹*Some hhf* = *Some h* $\oplus$ *Some hf*› *calculation* **by** *fastforce*

then obtain π' *uargs msf* **where** ($\forall$ *k*. *get-gu hf k* = *Some* (*uargs k*)) $\wedge$ ((π = *pwrite* $\wedge$ *get-gs hf* = *None* $\wedge$ *msf* = {#}) $\vee$ (*pwrite* = *padd* π π' $\wedge$ *get-gs hf* = *Some* (π', *msf*)))
by *blast*

then obtain *ghf* **where** *ghf-def*: *Some hf* = *Some* (*remove-guards hf*) $\oplus$ *Some ghf*
get-fh ghf = *Map.empty* (π = *pwrite* $\wedge$ *get-gs ghf* = *None* $\wedge$ *msf* = {#}) $\vee$ (*padd* π π' = *pwrite* $\wedge$ *get-gs ghf* = *Some* (π', *msf*))
$\bigwedge$*i*. *get-gu ghf i* = *Some* (*uargs i*)
using *decompose-guard-remove-easy*[*of hf*]
by (*metis fst-conv get-fh.elims get-gs.elims get-gu.simps snd-conv*)

have (*Map.empty*, *Some* (π, *?ms s'*), (λ-. *None*)) ## *ghf*
proof (*rule compatibleI*)
show *compatible-fract-heaps* (*get-fh* (*Map.empty*, *Some* (π, *?ms s'*), (λ-. *None*))) (*get-fh ghf*)
using *compatible-fract-heapsI* **by** *fastforce*
show $\bigwedge$*k*. *get-gu* (*Map.empty*, *Some* (π, {#*map-to-arg* (*s' sarg*)#} + *map-to-multiset* (*s' ms*)), *Map.empty*) *k* = *None* $\vee$ *get-gu ghf k* = *None*
by *simp*
fix *p p'*
assume *get-gs* (*Map.empty*, *Some* (π, {#*map-to-arg* (*s' sarg*)#} + *map-to-multiset* (*s' ms*)), *Map.empty*) = *Some p* $\wedge$ *get-gs ghf* = *Some p'*
then have *p* = (π, *?ms s'*) $\wedge$ *p'* = (π', *msf*) $\wedge$ *padd* π π' = *pwrite*
using *ghf-def* **by** *auto*
then show *pgte pwrite* (*padd* (*fst p*) (*fst p'*))
using *not-pgte-charact pgt-implies-pgte* **by** *auto*
qed
then obtain *g* **where** *g-def*: *Some g* = *Some* (*Map.empty*, *Some* (π, *?ms s'*), (λ-. *None*)) $\oplus$ *Some ghf*
by *simp*
moreover have *H'* ## *g*
proof (*rule compatibleI*)
have *get-fh g* = *add-fh Map.empty Map.empty* **using** *add-get-fh*[*of g* (*Map.empty*, *Some* (π, *?ms s'*), (λ-. *None*)) *ghf*]
g-def ‹*get-fh ghf* = *Map.empty*›
by *fastforce*
then have *get-fh g* = *Map.empty*
using *add-fh-map-empty* **by** *auto*
then show *compatible-fract-heaps* (*get-fh H'*) (*get-fh g*)
using *compatible-fract-heapsI* **by** *force*
show $\bigwedge$*k*. *get-gu H' k* = *None* $\vee$ *get-gu g k* = *None*
by (*meson asm3*(*4*) *no-guard-def*)
show $\bigwedge$*p p'*. *get-gs H'* = *Some p* $\wedge$ *get-gs g* = *Some p'* $\Longrightarrow$ *pgte pwrite*

(*padd* (*fst p*) (*fst p'*))
by (*metis asm3*(*4*) *no-guard-def option.simps*(*3*))
qed
then obtain H'' **where** *Some* H'' = *Some* H' ⊕ *Some g*
by *simp*
then have *Some* H'' = *Some* (*add-sguard-to-no-guard hq'* π (*?ms s'*)) ⊕ *Some hj'* ⊕ *Some hf*
proof −
have *Some* H'' = *Some h1* ⊕ *Some g* ⊕ *Some* (*remove-guards hf*)
by (*metis* ‹*Some* H'' = *Some* H' ⊕ *Some g*› *asm3*(*1*) *plus-comm simpler-asso*)
moreover have *Some* (*add-sguard-to-no-guard hq'* π (*?ms s'*)) = *Some hq'* ⊕ *Some* (*Map.empty*, *Some* (π, *?ms s'*), (λ-. *None*))
using ‹*Some* (*add-sguard-to-no-guard hq'* π ({#*map-to-arg* (*s' sarg*)#} + *map-to-multiset* (*s' ms*))) = *Some hq'* ⊕ *Some* (*Map.empty*, *Some* (π, {#*map-to-arg* (*s' sarg*)#} + *map-to-multiset* (*s' ms*)), (λ-. *None*))› **by** *blast*
ultimately show *?thesis*
by (*metis* (*no-types*, *lifting*) ‹*Some h1* = *Some hq'* ⊕ *Some hj'*› *g-def ghf-def*(*1*) *plus-comm simpler-asso*)
qed

moreover have *semi-consistent* Γ *v0* H''
proof (*rule semi-consistentI*)
have *get-gs g* = *Some* (*pwrite*, *?ms s'* + *msf*)
proof (*cases* π = *pwrite*)
case *True*
then have π = *pwrite* ∧ *get-gs ghf* = *None* ∧ *msf* = {#} **using** *ghf-def*(*3*)
by (*metis not-pgte-charact pgt-implies-pgte sum-larger*)
then show *?thesis*
by (*metis add.right-neutral fst-conv g-def get-gs.simps snd-conv sum-gs-one-none*)
next
case *False*
then have *padd* π π' = *pwrite* ∧ *get-gs ghf* = *Some* (π', *msf*)
using *ghf-def*(*3*) **by** *blast*
then show *?thesis*
by (*metis calculation*(*2*) *fst-conv get-gs.elims snd-conv sum-gs-one-some*)
qed

moreover have ⋀*i*. *get-gu g i* = *Some* (*uargs i*)
by (*metis full-uguard-sum-same ghf-def*(*4*) *g-def plus-comm*)
ultimately have *all-guards g*
using *all-guards-def* **by** *blast*
then show *all-guards* H''
by (*metis* ‹*Some* H'' = *Some* H' ⊕ *Some g*› *all-guards-same plus-comm*)
show *reachable* Γ *v0* H''

proof (*rule reachableI*)
fix *sargs uargs'*
assume *get-gs H'' = Some (pwrite, sargs) ∧ (∀ k. get-gu H'' k = Some (uargs' k))*
then have *sargs = ?ms s' + msf*
by (*metis (no-types, opaque-lifting) ‹Some H'' = Some H' ⊕ Some g› ‹get-gs g = Some (pwrite, {#map-to-arg (s' sarg)#} + map-to-multiset (s' ms) + msf)› asm3(4) no-guard-remove(1) option.inject plus-comm snd-conv*)
moreover have *uargs = uargs'*
apply (*rule ext*)
by (*metis ‹Some H'' = Some H' ⊕ Some g› ‹⋀i. get-gu g i = Some (uargs i)› ‹get-gs H'' = Some (pwrite, sargs) ∧ (∀ k. get-gu H'' k = Some (uargs' k))› asm3(4) no-guard-remove(2) option.sel plus-comm*)
moreover have *view Γ (FractionalHeap.normalize (get-fh hj')) = view Γ (FractionalHeap.normalize (get-fh H''))*
using *assms(4) ‹sat-inv s' hj' Γ›*
proof (*rule view-function-of-invE*)
show *H'' ⪰ hj'*
by (*metis (no-types, opaque-lifting) ‹Some H'' = Some H' ⊕ Some g› ‹Some h1 = Some hq' ⊕ Some hj'› asm3(1) larger-def larger-trans plus-comm*)
qed
moreover have *reachable-value (saction Γ) (uaction Γ) v0 (?ms s' + msf) uargs (sact v (map-to-arg (s' sarg)))*
proof −

have *reachable-value (saction Γ) (uaction Γ) v0 (?pre-ms s + msf) uargs (view Γ (FractionalHeap.normalize (get-fh H)))*
proof −
have *reachable Γ v0 H*
by (*meson asm2 semi-consistent-def*)
moreover have *get-gs H = Some (pwrite, ?pre-ms s + msf) ∧ (∀ k. get-gu H k = Some (uargs k))*
proof (*rule conjI*)
show *∀ k. get-gu H k = Some (uargs k)*
by (*metis ‹Some H = Some hhj ⊕ Some hf› full-uguard-sum-same ghf-def(1) ghf-def(4) plus-comm*)

moreover have *get-gs hhj = Some (π, ?pre-ms s)*
proof −
have *get-gs hj = None*
using *asm2 no-guard-def sat-inv-def* **by** *blast*
moreover have *get-gs h = Some (π, ?pre-ms s)*
using *‹get-gu h = Map.empty ∧ get-gs h = Some (π, map-to-multiset (s ms))›* **by** *blast*
ultimately show *?thesis*
by (*metis ‹Some hhj = Some h ⊕ Some hj› sum-gs-one-none*)
qed
ultimately show *get-gs H = Some (pwrite, ?pre-ms s + msf)*
proof (*cases π = pwrite*)

case *True*
then have π = *pwrite* $\wedge$ *get-gs ghf* = *None* $\wedge$ *msf* = {#} **using** *ghf-def*(*3*)
by (*metis not-pgte-charact pgt-implies-pgte sum-larger*)
then show *?thesis*
by (*metis* ‹*Some H* = *Some hhj* $\oplus$ *Some hf*› ‹*get-gs hhj* = *Some* (π, *map-to-multiset* (*s ms*))› *add.right-neutral full-sguard-sum-same*)
next
case *False*
then have *padd* π π' = *pwrite* $\wedge$ *get-gs ghf* = *Some* (π', *msf*)
using *ghf-def*(*3*) **by** *blast*
then show *?thesis* **using** ‹*Some H* = *Some hhj* $\oplus$ *Some hf*› *sum-gs-one-some ghf-def*(*1*)
‹*get-gs hhj* = *Some* (π, *?pre-ms s*)› *asm3*(*1*) *asm3*(*4*) *no-guard-remove*(*1*)[*of hf ghf remove-guards hf*] *no-guard-then-smaller-same plus-comm*
by *metis*
qed
qed
ultimately show *?thesis*
by (*meson reachableE*)
qed
moreover have *view* Γ (*FractionalHeap.normalize* (*get-fh hj*)) = *view* Γ (*FractionalHeap.normalize* (*get-fh H*))
using *assms*(*4*)
proof (*rule view-function-of-invE*)
show *sat-inv s hj* Γ
by (*simp add*: *asm2-bis*)
show *H* $\succeq$ *hj*
by (*metis* (*no-types*, *opaque-lifting*) ‹*Some H* = *Some hhj* $\oplus$ *Some hf*› ‹*Some hhj* = *Some h* $\oplus$ *Some hj*› *larger-def larger-trans plus-comm*)
qed
ultimately have *reachable-value* (*saction* Γ) (*uaction* Γ) *v0* (*?pre-ms s* + *msf*) *uargs v*
using ‹*f* (*FractionalHeap.normalize* (*get-fh hj*)) = *v*› *assms*(*1*) **by** *auto*
then show *?thesis*
using *SharedStep assms*(*1*)
using ‹*s x* = *s' x* $\wedge$ *s sarg* = *s' sarg* $\wedge$ *s ms* = *s' ms*› **by** *fastforce*
qed
ultimately show *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs uargs'* (*view* Γ (*FractionalHeap.normalize* (*get-fh H''*)))
using ‹*f* (*FractionalHeap.normalize* (*get-fh hj'*)) = *sact v* (*map-to-arg* (*s' sarg*))› *assms*(*1*) **by** *force*
qed
qed
ultimately show $\exists$ *H''*. *semi-consistent* Γ *v0 H''* $\wedge$ *Some H''* = *Some* (*add-sguard-to-no-guard hq'* π ({#*map-to-arg* (*s' sarg*)#} + *map-to-multiset* (*s' ms*))) $\oplus$ *Some hj'* $\oplus$ *Some hf*
by *blast*

qed
ultimately obtain H'' **where** *semi-consistent* Γ *v0* $H'' \wedge$ *Some* H'' = *Some* (*add-sguard-to-no-guard hq'* π (*?ms s'*)) $\oplus$ *Some hj'* $\oplus$ *Some hf*
$\wedge$ *safe n* (*Some* Γ) C' (s', *add-sguard-to-no-guard hq'* π (*?ms s'*)) (Σ' (*pre-s*, *h*)) **by** *blast*
moreover have *full-ownership* (*get-fh* H'') $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H'')
proof −
obtain *x* **where** *Some x* = *Some* (*add-sguard-to-no-guard hq'* π (*?ms s'*)) $\oplus$ *Some hj'*
by (*metis calculation not-Some-eq plus.simps*(*1*))
then have *get-fh* H'' = *add-fh* (*add-fh* (*get-fh* (*add-sguard-to-no-guard hq'* π (*?ms s'*))) (*get-fh hj'*)) (*get-fh hf*)
by (*metis add-get-fh calculation*)
moreover have *get-fh* (*add-sguard-to-no-guard hq'* π (*?ms s'*)) = *get-fh hq'* $\wedge$ *get-fh hf* = *get-fh* (*remove-guards hf*)
by (*metis get-fh-add-sguard get-fh-remove-guards*)
ultimately show *?thesis*
by (*metis* ‹*Some h1* = *Some hq'* $\oplus$ *Some hj'*› *add-get-fh asm3*(*1*) *asm3*(*3*) *asm3*(*4*))
qed
moreover have *sat-inv pre-s' hj'* Γ
proof (*rule sat-inv-agrees*)
show *sat-inv s' hj'* Γ
by (*simp add*: ‹*sat-inv s' hj'* Γ›)
show *agrees* (*fvA* (*invariant* Γ)) *s' pre-s'*
using *UnCI* ‹*agrees* (− {*x*}) *s' pre-s'*› *assms*(*1*) *assms*(*5*) *select-convs*(*5*) *subset-Compl-singleton*
by (*metis* (*mono-tags*, *lifting*) *agrees-def in-mono*)
qed
moreover have *safe n* (*Some* Γ) C' (*pre-s'*, *add-sguard-to-no-guard hq'* π (*?ms s'*)) (*?*Σ' (*pre-s*, *h*))
proof (*rule safe-free-vars-Some*)
show *safe n* (*Some* Γ) C' (s', *add-sguard-to-no-guard hq'* π (*?ms s'*)) (*?*Σ' (*pre-s*, *h*))
by (*meson* ‹*safe n* (*Some* Γ) C' (s', *add-sguard-to-no-guard hq'* π ({#*map-to-arg* (*s' sarg*)#} + *map-to-multiset* (*s' ms*))) (Σ' (*pre-s*, *h*))› *close-var-subset safe-larger-set*)
show *agrees* (*fvC* C' $\cup$ (− {*x*}) $\cup$ *fvA* (*invariant* Γ)) *s' pre-s'*
by (*metis UnI2 Un-absorb1* ‹*agrees* (− {*x*}) *s' pre-s'*› *asm3*(*2*) *assms*(*1*) *assms*(*5*) *empty-iff fvC.simps*(*1*) *inf-sup-aci*(*5*) *select-convs*(*5*) *subset-Compl-singleton*)
show *upper-fvs* (*close-var* (Σ' (*pre-s*, *h*)) *x*) (− {*x*})
by (*simp add*: *upper-fvs-close-vars*)
qed
ultimately show $\exists h''\ H'\ hj'$.
full-ownership (*get-fh* H') $\wedge$
semi-consistent Γ *v0* H' $\wedge$
sat-inv pre-s' hj' Γ $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some hj'* $\oplus$ *Some hf*

$\wedge$ *safe n* (*Some* Γ) C' (*pre-s'*, h'') (*?Σ'* (*pre-s*, h)) **using** ‹*sat-inv s' hj'* Γ›
by *blast*
qed (*simp*)
ultimately show *safe k* (*Some* Γ) (*Catomic C*) (*pre-s*, h) (*?Σ'* (*pre-s*, h))
by *blast*
qed (*simp*)
qed
qed

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
using *assms*(*1*)
apply (*rule red.cases*)
apply *blast+*
apply (*simp add*: *assms*(*2*))
apply (*simp add*: *assms*(*3*))
apply (*simp add*: *assms*(*4*))
apply *blast+*
done

lemma *no-abort-par*:
assumes *no-abort* Γ *C1 s h*
and *no-abort* Γ *C2 s h*
and *safe* (*Suc n*) Δ *C1* (*s*, *h1*) *S1*
and *safe* (*Suc n*) Δ *C2* (*s*, *h2*) *S2*
and *Some h* = *Some h1* $\oplus$ *Some h2*
and *bounded h*
shows *no-abort* Γ (*Cpar C1 C2*) *s h* $\wedge$ *accesses* (*Cpar C1 C2*) *s* $\subseteq$ *dom* (*fst h*)
$\wedge$ *writes* (*Cpar C1 C2*) *s* $\subseteq$ *fpdom* (*fst h*)
proof (*rule conjI*)
have *r*: *accesses C1 s* $\subseteq$ *dom* (*fst h1*) $\wedge$ *accesses C2 s* $\subseteq$ *dom* (*fst h2*) $\wedge$ *writes C1 s* $\subseteq$ *fpdom* (*fst h1*) $\wedge$ *writes C2 s* $\subseteq$ *fpdom* (*fst h2*)
using *assms*(*3*−*4*)
by (*metis fst-eqD safeAccessesE snd-eqD*)
then have *accesses* (*Cpar C1 C2*) *s* $\subseteq$ *dom* (*fst h*)
by (*metis Un-mono accesses.simps*(*8*) *assms*(*5*) *dom-sum-two get-fh.elims*)
moreover have *fpdom* (*fst h1*) $\cup$ *fpdom* (*fst h2*) $\subseteq$ *fpdom* (*fst h*)
using *assms*(*5*) *fpdom-dom-union assms*(*6*) **by** *blast*
then have *writes* (*Cpar C1 C2*) *s* $\subseteq$ *fpdom* (*fst h*)
using ‹*accesses C1 s* $\subseteq$ *dom* (*fst h1*) $\wedge$ *accesses C2 s* $\subseteq$ *dom* (*fst h2*) $\wedge$ *writes C1 s* $\subseteq$ *fpdom* (*fst h1*) $\wedge$ *writes C2 s* $\subseteq$ *fpdom* (*fst h2*)› *dual-order.trans* **by** *auto*
ultimately show *accesses* (*Cpar C1 C2*) *s* $\subseteq$ *dom* (*fst h*) $\wedge$ *writes* (*Cpar C1*

C2) $s \subseteq$ *fpdom* (*fst h*) **by** *simp*

have *r*: *accesses C1 s* $\cap$ *writes C2 s* = {} $\wedge$ *accesses C2 s* $\cap$ *writes C1 s* = {}
apply (*rule conjI*)
using *fpdom-dom-disjoint*[*OF assms*(*5*)] *r*
apply *blast*
using *assms*(*5*) *fpdom-dom-disjoint*[*of h h2 h1*] *r plus-comm*[*of Some h1 Some h2*]
by *force*

show *no-abort* Γ (*Cpar C1 C2*) *s h*
proof (*rule no-abortI*)
show $\bigwedge$*hf H*.
Some H = *Some h* $\oplus$ *Some hf* $\wedge$ Γ = *None* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H* $\Longrightarrow$
$\neg$ *aborts* (*Cpar C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*))
proof –
fix *hf H* **assume** *asm0*: *Some H* = *Some h* $\oplus$ *Some hf* $\wedge$ Γ = *None* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H*
let *?H* = *FractionalHeap.normalize* (*get-fh H*)

show $\neg$ *aborts* (*Cpar C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*))
proof (*rule ccontr*)
assume $\neg$ $\neg$ *aborts* (*Cpar C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*))
then have *aborts* (*Cpar C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) **by** *simp*
then have *aborts C1* (*s*, *?H*) $\vee$ *aborts C2* (*s*, *?H*)
apply (*rule aborts.cases*)
apply *simp-all*
using *r* **by** *force+*
then show *False*
using *asm0 assms*(*1*) *assms*(*2*) *no-abortE*(*1*) **by** *blast*
qed
qed
fix *H hf hj v0* Γ'
assume *asm0*: Γ = *Some* Γ' $\wedge$ *Some H* = *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *semi-consistent* Γ' *v0 H* $\wedge$ *sat-inv s hj* Γ'
let *?H* = *FractionalHeap.normalize* (*get-fh H*)
show $\neg$ *aborts* (*Cpar C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*))
proof (*rule ccontr*)
assume $\neg$ $\neg$ *aborts* (*Cpar C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*))
then have *aborts* (*Cpar C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) **by** *simp*
then have *aborts C1* (*s*, *?H*) $\vee$ *aborts C2* (*s*, *?H*)
apply (*rule aborts.cases*)
apply *simp-all*
using *r* **by** *force+*
then show *False*
using *asm0 assms*(*1*) *assms*(*2*) *no-abortE*(*2*) **by** *blast*

qed
qed
qed

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* ⊕ *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 *bounded h*

shows *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) (*Cpar C1 C2*) (*s*, *h*) (*add-states S1 S2*)
using *assms*
proof (*induct n arbitrary*: *C1 h1 C2 h2 s h S1 S2*)
case (*Suc n*)
show *?case*
proof (*rule safeNoneI*)
show *Cpar C1 C2* = *Cskip* ⟹ (*s*, *h*) ∈ *add-states S1 S2*
by *simp*
have *r*: *no-abort* (*None* :: (*'i*, *'a*, *nat*) *cont*) (*Cpar C1 C2*) *s h* ∧ *accesses* (*Cpar C1 C2*) *s* ⊆ *dom* (*fst h*) ∧ *writes* (*Cpar C1 C2*) *s* ⊆ *fpdom* (*fst h*)
proof (*rule no-abort-par*)
show *no-abort* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C1 s h*
using *Suc.prems*(*1*) *Suc.prems*(*3*) *larger-def no-abort-larger safe.simps*(*2*) **by** *blast*
have *h* ⪰ *h2*
by (*metis Suc.prems*(*3*) *larger-def plus-comm*)
then show *no-abort* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C2 s h*
using *Suc.prems*(*2*) *no-abort-larger safeNoneE-bis*(*2*) **by** *blast*

show *safe* (*Suc n*) *None C1* (*s*, *h1*) *S1* **using** *Suc*(*2*) **by** *simp*
show *safe* (*Suc n*) *None C2* (*s*, *h2*) *S2* **using** *Suc*(*3*) **by** *simp*
qed (*simp-all add*: *Suc*)
then show *no-abort* (*None* :: (*'i*, *'a*, *nat*) *cont*) (*Cpar C1 C2*) *s h* **by** *simp*
show *accesses* (*Cpar C1 C2*) *s* ⊆ *dom* (*fst h*) ∧ *writes* (*Cpar C1 C2*) *s* ⊆ *fpdom* (*fst h*) **using** *r* **by** *simp*

fix *H hf C' s' h'*
assume *asm0*: *Some H* = *Some h* ⊕ *Some hf* ∧
full-ownership (*get-fh H*) ∧ *no-guard H* ∧ *red* (*Cpar C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C'* (*s'*, *h'*)

obtain *hf1* **where** *Some hf1 = Some h1 ⊕ Some hf*
by (*metis* (*no-types*, *opaque-lifting*) *Suc.prems*(*3*) *asm0 plus.simps*(*1*) *plus.simps*(*3*) *plus-asso plus-comm*)
then have *Some H = Some h2 ⊕ Some hf1*
by (*metis* (*no-types*, *lifting*) *Suc.prems*(*3*) *asm0 plus-asso plus-comm*)
obtain *hf2* **where** *Some hf2 = Some h2 ⊕ Some hf*
by (*metis* (*no-types*, *opaque-lifting*) ‹*Some H = Some h2 ⊕ Some hf1*› ‹*Some hf1 = Some h1 ⊕ Some hf*› *option.exhaust-sel plus.simps*(*1*) *plus-asso plus-comm*)
then have *Some H = Some h1 ⊕ Some hf2*
by (*metis Suc.prems*(*3*) *asm0 plus-asso*)

let *?H = normalize* (*get-fh H*)

show $\exists h''\ H'$.
full-ownership (*get-fh H'*) ∧
no-guard H' ∧ *h' = FractionalHeap.normalize* (*get-fh H'*) ∧ *Some H' = Some h'' ⊕ Some hf* ∧ *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C'* (*s'*, *h''*) (*add-states S1 S2*)
proof (*rule par-cases*)
show *red* (*Cpar C1 C2*) (*s*, *?H*) *C'* (*s'*, *h'*)
using *asm0* **by** *blast*

show *C1 = Cskip* ∧ *C2 = Cskip* ∧ *C' = Cskip* ∧ (*s*, *FractionalHeap.normalize* (*get-fh H*)) = (*s'*, *h'*) ⟹
$\exists h''\ H'$. *full-ownership* (*get-fh H'*) ∧
no-guard H' ∧ *h' = FractionalHeap.normalize* (*get-fh H'*) ∧ *Some H' = Some h'' ⊕ Some hf* ∧ *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C'* (*s'*, *h''*) (*add-states S1 S2*)
proof −
assume *asm1*: *C1 = Cskip* ∧ *C2 = Cskip* ∧ *C' = Cskip* ∧ (*s*, *FractionalHeap.normalize* (*get-fh H*)) = (*s'*, *h'*)
then have (*s*, *h1*) ∈ *S1* ∧ (*s*, *h2*) ∈ *S2*
using *Suc.prems*(*1*) *Suc.prems*(*2*) *safe.simps*(*2*) **by** *blast*
moreover have (*s*, *h*) ∈ *add-states S1 S2*
by (*metis* (*mono-tags*, *lifting*) *Suc.prems*(*3*) *add-states-def calculation mem-Collect-eq*)
ultimately show $\exists h''\ H'$.
full-ownership (*get-fh H'*) ∧
no-guard H' ∧ *h' = FractionalHeap.normalize* (*get-fh H'*) ∧ *Some H' = Some h'' ⊕ Some hf* ∧ *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C'* (*s'*, *h''*) (*add-states S1 S2*)
by (*metis asm0 asm1 old.prod.inject safe-skip*)
qed

show $\bigwedge$*C1'*. *C' = Cpar C1' C2* ∧ *red C1* (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C1'* (*s'*, *h'*) ⟹
$\exists h''\ H'$.
full-ownership (*get-fh H'*) ∧
no-guard H' ∧ *h' = FractionalHeap.normalize* (*get-fh H'*) ∧ *Some H'*

= *Some h''* ⊕ *Some hf* ∧ *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C'* (*s'*, *h''*) (*add-states S1 S2*)
proof −
fix *C1'*
assume *asm1*: *C'* = *Cpar C1' C2* ∧ *red C1* (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C1'* (*s'*, *h'*)
then obtain *h1' H'* **where** *asm2*: *full-ownership* (*get-fh H'*) *no-guard H'* *h'* = *FractionalHeap.normalize* (*get-fh H'*)
Some H' = *Some h1'* ⊕ *Some hf2* *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C1'* (*s'*, *h1'*) *S1*
using *Suc.prems*(*1*) *asm0* *safeNoneE*(*3*)[*of n C1 s h1 S1 H hf2 C1' s' h'*] ‹*Some H* = *Some h1* ⊕ *Some hf2*› **by** *blast*

moreover have *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C2* (*s*, *h2*) *S2*
by (*meson Suc.prems*(*2*) *Suc-leD le-Suc-eq safe-smaller*)

then have *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C2* (*s'*, *h2*) *S2*
proof (*rule safe-free-vars-None*)
show *agrees* (*fvC C2* ∪ *vars2*) *s s'*
using *Suc.prems*(*5*) *agrees-minusD*[*of*] *agrees-comm asm1 fst-eqD red-properties*(*1*) *disjoint-def inf-commute*
by *metis*
show *upper-fvs S2 vars2*
by (*simp add*: *Suc.prems*(*7*))
qed

moreover obtain *h''* **where** *Some h''* = *Some h1'* ⊕ *Some h2*
by (*metis* ‹*Some hf2* = *Some h2* ⊕ *Some hf*› *calculation*(*4*) *not-Some-eq plus.simps*(*1*) *plus-asso*)
have *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) (*Cpar C1' C2*) (*s'*, *h''*) (*add-states S1 S2*)
proof (*rule Suc.hyps*)
show *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C1'* (*s'*, *h1'*) *S1*
using *calculation*(*5*) **by** *blast*
show *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C2* (*s'*, *h2*) *S2*
using *calculation*(*6*) **by** *auto*
show *Some h''* = *Some h1'* ⊕ *Some h2*
using ‹*Some h''* = *Some h1'* ⊕ *Some h2*› **by** *blast*
show *disjoint* (*fvC C1'* ∪ *vars1*) (*wrC C2*)
using *Suc.prems*(*4*) *asm1 red-properties*(*1*) *Un-iff disjoint-def*[*of fvC C1* ∪ *vars1 wrC C2*]
disjoint-def[*of fvC C1'* ∪ *vars1 wrC C2*]
inf-shunt subset-iff **by** *blast*
show *disjoint* (*fvC C2* ∪ *vars2*) (*wrC C1'*)
by (*metis* (*no-types*, *lifting*) *Suc.prems*(*5*) *asm1 disjoint-def inf-commute inf-shunt red-properties*(*1*) *subset-Un-eq sup-assoc*)
show *upper-fvs S1 vars1*
by (*simp add*: *Suc.prems*(*6*))
show *upper-fvs S2 vars2*

by (*simp add*: *Suc.prems*(*7*))
show *bounded* h''
apply (*rule bounded-smaller*[*of* H'])
using *calculation*(*1*) *full-ownership-then-bounded* **apply** *fastforce*
by (*metis* ‹*Some* h'' = *Some* $h1'$ ⊕ *Some* $h2$› ‹*Some* $hf2$ = *Some* $h2$ ⊕ *Some* hf› *calculation*(*4*) *larger-def plus-asso*)
qed

ultimately show $\exists h''\ H'$.
full-ownership (*get-fh* H') ∧
no-guard H' ∧
h' = *FractionalHeap.normalize* (*get-fh* H') ∧ *Some* H' = *Some* h'' ⊕ *Some hf* ∧ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (s', h'') (*add-states S1 S2*)
by (*metis* ‹*Some* h'' = *Some* $h1'$ ⊕ *Some* $h2$› ‹*Some* $hf2$ = *Some* $h2$ ⊕ *Some* hf› *asm1 plus-asso*)
qed
show ⋀$C2'$. C' = *Cpar C1* $C2'$ ∧ *red C2* (*s*, *FractionalHeap.normalize* (*get-fh H*)) $C2'$ (s', h') ⟹
$\exists h''\ H'$.
full-ownership (*get-fh* H') ∧
no-guard H' ∧ h' = *FractionalHeap.normalize* (*get-fh* H') ∧ *Some* H' = *Some* h'' ⊕ *Some hf* ∧ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (s', h'') (*add-states S1 S2*)
proof −
fix $C2'$
assume *asm1*: C' = *Cpar C1* $C2'$ ∧ *red C2* (*s*, *FractionalHeap.normalize* (*get-fh H*)) $C2'$ (s', h')
then obtain $h2'$ H' **where** *asm2*: *full-ownership* (*get-fh* H') *no-guard* H' h' = *FractionalHeap.normalize* (*get-fh* H')
Some H' = *Some* $h2'$ ⊕ *Some hf1* *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) $C2'$ (s', $h2'$) *S2*
using *Suc.prems*(*1*) *asm0 safeNoneE*(*3*) *Suc.prems*(*2*) ‹*Some H* = *Some h2* ⊕ *Some hf1*› **by** *blast*

moreover have *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C1* (*s*, *h1*) *S1*
by (*meson Suc.prems*(*1*) *Suc-leD le-Suc-eq safe-smaller*)

then have *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C1* (s', *h1*) *S1*
proof (*rule safe-free-vars-None*)
show *agrees* (*fvC C1* ∪ *vars1*) *s* s'
using *Suc.prems*(*4*) *agrees-comm asm1 fst-eqD red-properties*(*1*) *disjoint-def*[*of fvC C1* ∪ *vars1 wrC C2*]
agrees-minusD **by** (*metis inf-commute*)
show *upper-fvs S1 vars1*
by (*simp add*: *Suc.prems*(*6*))
qed

moreover obtain h'' **where** *Some* h'' = *Some* $h2'$ ⊕ *Some h1*
by (*metis* ‹*Some hf1* = *Some h1* ⊕ *Some hf*› *calculation*(*4*) *not-Some-eq*

```
plus.simps(1) plus-asso)
      have safe n (None :: ('i, 'a, nat) cont) (Cpar C1 C2') (s', h'') (add-states
S1 S2)
      proof (rule Suc.hyps)
        show safe n (None :: ('i, 'a, nat) cont) C1 (s', h1) S1
          using calculation(6) by blast
        show safe n (None :: ('i, 'a, nat) cont) C2' (s', h2') S2
          using calculation(5) by auto
        show Some h'' = Some h1 ⊕ Some h2'
          by (simp add: ‹Some h'' = Some h2' ⊕ Some h1› plus-comm)
        show disjoint (fvC C2' ∪ vars2) (wrC C1)
            using Suc.prems(5) asm1 disjoint-def[of fvC C2 ∪ vars2 wrC C1]
disjoint-def[of fvC C2' ∪ vars2 wrC C1]
         inf-shunt inf-sup-aci(5) red-properties(1) subset-Un-eq sup.idem sup-assoc
          by fast
        show disjoint (fvC C1 ∪ vars1) (wrC C2')
         by (metis (no-types, lifting) Suc.prems(4) asm1 disjoint-def inf-commute
inf-shunt red-properties(1) subset-Un-eq sup-assoc)
        show upper-fvs S1 vars1
          by (simp add: Suc.prems(6))
        show upper-fvs S2 vars2
          by (simp add: Suc.prems(7))
        show bounded h''
          apply (rule bounded-smaller[of H'])
          using calculation(1) full-ownership-then-bounded apply fastforce
          by (metis ‹Some h'' = Some h2' ⊕ Some h1› ‹Some hf1 = Some h1 ⊕
Some hf› calculation(4) larger-def plus-asso)
      qed

      ultimately show ∃h'' H'.
           full-ownership (get-fh H') ∧
           no-guard H' ∧
            h' = FractionalHeap.normalize (get-fh H') ∧ Some H' = Some h'' ⊕
Some hf ∧ safe n (None :: ('i, 'a, nat) cont) C' (s', h'') (add-states S1 S2)
         by (metis ‹Some h'' = Some h2' ⊕ Some h1› ‹Some hf1 = Some h1 ⊕
Some hf› asm1 plus-asso)
    qed
  qed
 qed
qed (simp)

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*)

and *bounded h*

shows *safe n* (*Some* Γ) (*Cpar C1 C2*) (*s*, *h*) (*add-states S1 S2*)
using *assms*
proof (*induct n arbitrary*: *C1 h1 C2 h2 s h S1 S2*)
case (*Suc n*)
show *?case*
proof (*rule safeSomeI*)
show *Cpar C1 C2* = *Cskip* ⟹ (*s*, *h*) ∈ *add-states S1 S2*
by *simp*
have *r*: *no-abort* (*Some* Γ) (*Cpar C1 C2*) *s h* ∧ *accesses* (*Cpar C1 C2*) *s* ⊆ *dom* (*fst h*) ∧ *writes* (*Cpar C1 C2*) *s* ⊆ *fpdom* (*fst h*)
proof (*rule no-abort-par*)
show *no-abort* (*Some* Γ) *C1 s h*
using *Suc.prems*(*1*) *Suc.prems*(*3*) *larger-def no-abort-larger safe.simps*(*3*) **by** *blast*
have *h* ⪰ *h2*
by (*metis Suc.prems*(*3*) *larger-def plus-comm*)
then show *no-abort* (*Some* Γ) *C2 s h*
using *Suc.prems*(*2*) *no-abort-larger safeSomeE*(*2*) **by** *blast*

show *safe* (*Suc n*) (*Some* Γ) *C1* (*s*, *h1*) *S1* **using** *Suc*(*2*) **by** *simp*
show *safe* (*Suc n*) (*Some* Γ) *C2* (*s*, *h2*) *S2* **using** *Suc*(*3*) **by** *simp*
qed (*simp-all add*: *Suc*)

then show *accesses* (*Cpar C1 C2*) *s* ⊆ *dom* (*fst h*) ∧ *writes* (*Cpar C1 C2*) *s* ⊆ *fpdom* (*fst h*)
by *simp*
show *no-abort* (*Some* Γ) (*Cpar C1 C2*) *s h* **using** *r* **by** *simp*

fix *H hf C′ s′ h′ hj v0*
assume *asm0*: *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ∧ *full-ownership* (*get-fh H*) ∧
semi-consistent Γ *v0 H* ∧ *sat-inv s hj* Γ ∧ *red* (*Cpar C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)

obtain *hf1* **where** *Some hf1* = *Some h1* ⊕ *Some hf*
by (*metis* (*no-types*, *opaque-lifting*) *Suc.prems*(*3*) *asm0 plus.simps*(*1*) *plus.simps*(*3*) *plus-asso plus-comm*)
then have *Some H* = *Some h2* ⊕ *Some hf1* ⊕ *Some hj*
by (*metis* (*no-types*, *lifting*) *Suc.prems*(*3*) *asm0 plus-asso plus-comm*)
then have *Some H* = *Some h2* ⊕ *Some hj* ⊕ *Some hf1*

by (*metis plus-asso plus-comm*)
obtain *hf2* **where** *Some hf2* = *Some h2* ⊕ *Some hf*
by (*metis* (*no-types*, *opaque-lifting*) ‹*Some H* = *Some h2* ⊕ *Some hf1* ⊕ *Some hj*› ‹*Some hf1* = *Some h1* ⊕ *Some hf*› *not-Some-eq plus.simps*(*1*) *plus-asso plus-comm*)
then have *Some H* = *Some h1* ⊕ *Some hf2* ⊕ *Some hj*
by (*metis* (*no-types*, *opaque-lifting*) ‹*Some H* = *Some h2* ⊕ *Some hf1* ⊕ *Some hj*› ‹*Some hf1* = *Some h1* ⊕ *Some hf*› *plus-asso plus-comm*)
then have *Some H* = *Some h1* ⊕ *Some hj* ⊕ *Some hf2*
by (*metis plus-asso plus-comm*)

let *?H* = *normalize* (*get-fh H*)

show $\exists$ *h''* *H'* *hj'*.
full-ownership (*get-fh H'*) ∧
semi-consistent Γ *v0 H'* ∧
sat-inv s' hj' Γ ∧
h' = *FractionalHeap.normalize* (*get-fh H'*) ∧ *Some H'* = *Some h''* ⊕ *Some hj'* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C'* (*s'*, *h''*) (*add-states S1 S2*)
proof (*rule par-cases*)
show *red* (*Cpar C1 C2*) (*s*, *?H*) *C'* (*s'*, *h'*)
using *asm0* **by** *blast*

show *C1* = *Cskip* ∧ *C2* = *Cskip* ∧ *C'* = *Cskip* ∧ (*s*, *FractionalHeap.normalize* (*get-fh H*)) = (*s'*, *h'*) ⟹
$\exists$ *h''* *H'* *hj'*. *full-ownership* (*get-fh H'*) ∧
semi-consistent Γ *v0 H'* ∧
sat-inv s' hj' Γ ∧ *h'* = *FractionalHeap.normalize* (*get-fh H'*) ∧ *Some H'* = *Some h''* ⊕ *Some hj'* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C'* (*s'*, *h''*) (*add-states S1 S2*)
proof −
assume *asm1*: *C1* = *Cskip* ∧ *C2* = *Cskip* ∧ *C'* = *Cskip* ∧ (*s*, *FractionalHeap.normalize* (*get-fh H*)) = (*s'*, *h'*)
then have (*s*, *h1*) ∈ *S1* ∧ (*s*, *h2*) ∈ *S2*
using *Suc.prems*(*1*) *Suc.prems*(*2*) *safe.simps*(*3*) **by** *blast*
moreover have (*s*, *h*) ∈ *add-states S1 S2*
by (*metis* (*mono-tags*, *lifting*) *Suc.prems*(*3*) *add-states-def calculation mem-Collect-eq*)
ultimately show $\exists$ *h''* *H'* *hj'*. *full-ownership* (*get-fh H'*) ∧
semi-consistent Γ *v0 H'* ∧
sat-inv s' hj' Γ ∧ *h'* = *FractionalHeap.normalize* (*get-fh H'*) ∧ *Some H'* = *Some h''* ⊕ *Some hj'* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C'* (*s'*, *h''*) (*add-states S1 S2*)
by (*metis asm0 asm1 old.prod.inject safe-skip*)
qed

show ⋀*C1'*. *C'* = *Cpar C1' C2* ∧ *red C1* (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C1'* (*s'*, *h'*) ⟹
$\exists$ *h''* *H'* *hj'*. *full-ownership* (*get-fh H'*) ∧ *semi-consistent* Γ *v0 H'* ∧
sat-inv s' hj' Γ ∧ *h'* = *FractionalHeap.normalize* (*get-fh H'*) ∧ *Some H'* = *Some h''* ⊕ *Some hj'* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C'* (*s'*, *h''*) (*add-states*

S1 S2)
proof −
fix *C1'*
assume *asm1*: $C' = Cpar\ C1'\ C2 \wedge red\ C1\ (s,\ FractionalHeap.normalize\ (get\text{-}fh\ H))\ C1'\ (s',\ h')$
then obtain *h1' H' hj'* **where** *asm2*: $full\text{-}ownership\ (get\text{-}fh\ H')\ h' = FractionalHeap.normalize\ (get\text{-}fh\ H')$
$semi\text{-}consistent\ \Gamma\ v0\ H'\ sat\text{-}inv\ s'\ hj'\ \Gamma\ Some\ H' = Some\ h1' \oplus Some\ hj' \oplus Some\ hf2\ safe\ n\ (Some\ \Gamma)\ C1'\ (s',\ h1')\ S1$
using *safeSomeE*(*3*)[*of n* Γ *C1 s h1 S1 H hj hf2 v0 C1' s' h'*] *Suc.prems*(*1*) *asm0*
using ‹$Some\ H = Some\ h1 \oplus Some\ hj \oplus Some\ hf2$› **by** *blast*

moreover have *safe n* (*Some* Γ) *C2* (*s*, *h2*) *S2*
by (*meson Suc.prems*(*2*) *Suc-leD le-Suc-eq safe-smaller*)

then have *safe n* (*Some* Γ) *C2* (*s'*, *h2*) *S2*
proof (*rule safe-free-vars-Some*)
show $agrees\ (fvC\ C2 \cup vars2 \cup fvA\ (invariant\ \Gamma))\ s\ s'$
using *Suc.prems*(*5*) *Suc.prems*(*9*) *agrees-minusD agrees-comm asm1 disjoint-def fst-eqD red-properties*(*1*)
by (*metis agrees-union inf-commute*)
show *upper-fvs S2 vars2*
by (*simp add*: *Suc.prems*(*7*))
qed

moreover have *h1' ## h2*
by (*metis* (*no-types*, *opaque-lifting*) ‹$Some\ hf2 = Some\ h2 \oplus Some\ hf$› *calculation*(*5*) *compatible-eq option.discI plus.simps*(*1*) *plus-asso plus-comm*)
then obtain *h''* **where** $Some\ h'' = Some\ h1' \oplus Some\ h2$ **by** *simp*

have *safe n* (*Some* Γ) (*Cpar C1' C2*) (*s'*, *h''*) (*add-states S1 S2*)
proof (*rule Suc.hyps*)
show *safe n* (*Some* Γ) *C1'* (*s'*, *h1'*) *S1*
using *calculation*(*6*) **by** *blast*
show *safe n* (*Some* Γ) *C2* (*s'*, *h2*) *S2*
using *calculation*(*7*) **by** *auto*
show $Some\ h'' = Some\ h1' \oplus Some\ h2$
using ‹$Some\ h'' = Some\ h1' \oplus Some\ h2$› **by** *blast*
show $disjoint\ (fvC\ C1' \cup vars1)\ (wrC\ C2)$
by (*metis* (*no-types*, *opaque-lifting*) *Suc.prems*(*4*) *asm1 disjnt-Un1 disjnt-def disjoint-def red-properties*(*1*) *sup.orderE*)
show $disjoint\ (fvC\ C2 \cup vars2)\ (wrC\ C1')$
by (*metis* (*no-types*, *lifting*) *Suc.prems*(*5*) *asm1 disjoint-def inf-commute inf-shunt red-properties*(*1*) *subset-Un-eq sup-assoc*)
show *upper-fvs S1 vars1*
by (*simp add*: *Suc.prems*(*6*))
show *upper-fvs S2 vars2*
by (*simp add*: *Suc.prems*(*7*))

show *disjoint* (*fvA* (*invariant* Γ)) (*wrC C2*)
by (*simp add*: *Suc.prems*(*8*))
show *disjoint* (*fvA* (*invariant* Γ)) (*wrC C1′*)
by (*metis* (*no-types*, *lifting*) *Suc.prems*(*9*) *asm1 disjoint-def inf-commute inf-shunt red-properties*(*1*) *subset-Un-eq sup-assoc*)
show *bounded h′′*
apply (*rule bounded-smaller*[*of H′*])
using *calculation*(*1*) *full-ownership-then-bounded* **apply** *fastforce*
using ‹*Some h′′* = *Some h1′* ⊕ *Some h2*› ‹*Some hf2* = *Some h2* ⊕ *Some hf*› *calculation*(*5*) *larger3*[*of H′* - *h′′ hf*] *plus-asso plus-comm*[*of Some hj′*]
by *metis*

qed
moreover have *Some H′* = *Some h′′* ⊕ *Some hj′* ⊕ *Some hf*
by (*metis* (*no-types*, *opaque-lifting*) ‹*Some h′′* = *Some h1′* ⊕ *Some h2*› ‹*Some hf2* = *Some h2* ⊕ *Some hf*› *calculation*(*5*) *plus-asso plus-comm*)

ultimately show ∃ *h′′ H′ hj′*.
full-ownership (*get-fh H′*) ∧
semi-consistent Γ *v0 H′* ∧
sat-inv s′ hj′ Γ ∧ *h′* = *FractionalHeap.normalize* (*get-fh H′*) ∧ *Some H′* = *Some h′′* ⊕ *Some hj′* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C′* (*s′*, *h′′*) (*add-states S1 S2*)
using *asm1* **by** *blast*

qed

show ⋀*C2′*. *C′* = *Cpar C1 C2′* ∧ *red C2* (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C2′* (*s′*, *h′*) ⟹
∃ *h′′ H′ hj′*.
full-ownership (*get-fh H′*) ∧
semi-consistent Γ *v0 H′* ∧
sat-inv s′ hj′ Γ ∧ *h′* = *FractionalHeap.normalize* (*get-fh H′*) ∧ *Some H′* = *Some h′′* ⊕ *Some hj′* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C′* (*s′*, *h′′*) (*add-states S1 S2*)
proof −
fix *C2′*
assume *asm1*: *C′* = *Cpar C1 C2′* ∧ *red C2* (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C2′* (*s′*, *h′*)
then obtain *h2′ H′ hj′* **where** *asm2*: *full-ownership* (*get-fh H′*) *h′* = *FractionalHeap.normalize* (*get-fh H′*)
semi-consistent Γ *v0 H′ sat-inv s′ hj′* Γ *Some H′* = *Some h2′* ⊕ *Some hj′* ⊕ *Some hf1 safe n* (*Some* Γ) *C2′* (*s′*, *h2′*) *S2*
using *safeSomeE*(*3*)[*of n* Γ *C2 s h2 S2 H hj hf1 v0 C2′ s′ h′*] *Suc.prems*(*2*) *Suc.prems*(*3*)
using ‹*Some H* = *Some h2* ⊕ *Some hj* ⊕ *Some hf1*› *asm0* **by** *blast*
moreover have *safe n* (*Some* Γ) *C1* (*s*, *h1*) *S1*
by (*meson Suc.prems*(*1*) *Suc-leD le-Suc-eq safe-smaller*)

then have *safe n* (*Some* Γ) *C1* (*s*′, *h1*) *S1*
proof (*rule safe-free-vars-Some*)
show *agrees* (*fvC C1* ∪ *vars1* ∪ *fvA* (*invariant* Γ)) *s s*′
using *Suc.prems*(*4*) *Suc.prems*(*8*) *agrees-minusD agrees-comm asm1 fst-eqD red-properties*(*1*)
by (*metis agrees-union disjoint-def inf-commute*)
show *upper-fvs S1 vars1*
by (*simp add*: *Suc.prems*(*6*))
qed

moreover have *h1* ## *h2*′
by (*metis* (*no-types*, *opaque-lifting*) ‹*Some hf1* = *Some h1* ⊕ *Some hf*› *calculation*(*5*) *compatible-eq option.distinct*(*1*) *plus.simps*(*1*) *plus-asso plus-comm*)
then obtain *h*′′ **where** *Some h*′′ = *Some h1* ⊕ *Some h2*′ **by** *simp*

have *safe n* (*Some* Γ) (*Cpar C1 C2*′) (*s*′, *h*′′) (*add-states S1 S2*)
proof (*rule Suc.hyps*)
show *safe n* (*Some* Γ) *C1* (*s*′, *h1*) *S1*
using *calculation*(*7*) **by** *blast*
show *safe n* (*Some* Γ) *C2*′ (*s*′, *h2*′) *S2*
using *calculation*(*6*) **by** *auto*
show *Some h*′′ = *Some h1* ⊕ *Some h2*′
using ‹*Some h*′′ = *Some h1* ⊕ *Some h2*′› **by** *blast*
show *disjoint* (*fvC C1* ∪ *vars1*) (*wrC C2*′)
by (*metis* (*no-types*, *lifting*) *Suc.prems*(*4*) *asm1 disjoint-def inf-commute inf-shunt red-properties*(*1*) *subset-Un-eq sup-assoc*)
show *disjoint* (*fvC C2*′ ∪ *vars2*) (*wrC C1*)
using *Suc.prems*(*5*) *asm1 red-properties*(*1*)
by (*metis* (*no-types*, *lifting*) *Un-subset-iff disjoint-def inf-shunt subset-Un-eq*)
show *disjoint* (*fvA* (*invariant* Γ)) (*wrC C2*′)
using *Suc.prems*(*8*) *asm1 red-properties*(*1*)
by (*metis* (*no-types*, *lifting*) *Un-subset-iff disjoint-def inf-commute inf-shunt subset-Un-eq*)
show *disjoint* (*fvA* (*invariant* Γ)) (*wrC C1*)
by (*simp add*: *Suc.prems*(*9*))
show *upper-fvs S1 vars1*
by (*simp add*: *Suc.prems*(*6*))
show *upper-fvs S2 vars2*
by (*simp add*: *Suc.prems*(*7*))
show *bounded h*′′
apply (*rule bounded-smaller*[*of H*′])
using *calculation*(*1*) *full-ownership-then-bounded* **apply** *fastforce*
proof −
have *Some H*′ = *Some h1* ⊕ *Some h2*′ ⊕ *Some hj*′ ⊕ *Some hf*
by (*metis* (*no-types*, *lifting*) ‹*Some hf1* = *Some h1* ⊕ *Some hf*› *calculation*(*5*) *plus-asso plus-comm*)
then show *H*′ ⪰ *h*′′

by (*simp add*: ‹*Some h'' = Some h1* ⊕ *Some h2'*› *larger3 plus-comm*)
qed
qed
moreover have *Some H' = Some h''* ⊕ *Some hj'* ⊕ *Some hf*
by (*metis* ‹*Some h'' = Some h1* ⊕ *Some h2'*› ‹*Some hf1 = Some h1* ⊕ *Some hf*› *calculation*(*5*) *plus-comm simpler-asso*)
ultimately show ∃ *h'' H' hj'*.
full-ownership (*get-fh H'*) ∧
semi-consistent Γ *v0 H'* ∧
sat-inv s' hj' Γ ∧
h' = *FractionalHeap.normalize* (*get-fh H'*) ∧ *Some H' = Some h''* ⊕ *Some hj'* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C'* (*s'*, *h''*) (*add-states S1 S2*)
using *asm1* **by** *blast*
qed
qed
qed
qed (*simp*)

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*)

and *bounded h*

shows *safe n* Δ (*Cpar C1 C2*) (*s*, *h*) (*add-states S1 S2*)
proof (*cases* Δ)
case *None*
then show *?thesis*
using *assms parallel-comp-none* **by** *blast*
next
case (*Some* Γ)
then show *?thesis*
using *assms parallel-comp-some* **by** *blast*
qed

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* $\cup$ *fvC C1* $\cup$ *fvA Q1*) (*wrC C2*)
and *disjoint* (*fvA P2* $\cup$ *fvC C2* $\cup$ *fvA Q2*) (*wrC C1*)

and $\bigwedge\Gamma.\ \Delta = Some\ \Gamma \Longrightarrow disjoint\ (fvA\ (invariant\ \Gamma))\ (wrC\ C2)$
and $\bigwedge\Gamma.\ \Delta = Some\ \Gamma \Longrightarrow disjoint\ (fvA\ (invariant\ \Gamma))\ (wrC\ C1)$

and *precise P1* $\vee$ *precise P2*

shows *hoare-triple-valid* Δ (*Star P1 P2*) (*Cpar C1 C2*) (*Star Q1 Q2*)
proof –
obtain $\Sigma 1$ **where** *r1*: $\bigwedge\sigma\ n.\ \sigma, \sigma \models P1 \Longrightarrow bounded\ (snd\ \sigma) \Longrightarrow safe\ n\ \Delta\ C1\ \sigma\ (\Sigma 1\ \sigma)\ \bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models P1 \Longrightarrow pair\text{-}sat\ (\Sigma 1\ \sigma)\ (\Sigma 1\ \sigma')\ Q1$
using *assms*(*1*) *hoare-triple-validE* **by** *blast*
obtain $\Sigma 2$ **where** *r2*: $\bigwedge\sigma\ n.\ \sigma, \sigma \models P2 \Longrightarrow bounded\ (snd\ \sigma) \Longrightarrow safe\ n\ \Delta\ C2\ \sigma\ (\Sigma 2\ \sigma)\ \bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models P2 \Longrightarrow pair\text{-}sat\ (\Sigma 2\ \sigma)\ (\Sigma 2\ \sigma')\ Q2$
using *assms*(*2*) *hoare-triple-validE* **by** *blast*

define *pairs* **where** $pairs = (\lambda(s, h).\ \{\ ((s, h1), (s, h2))\ | h1\ h2.\ Some\ h = Some\ h1 \oplus Some\ h2 \wedge (s, h1), (s, h1) \models P1 \wedge (s, h2), (s, h2) \models P2\ \})$
define Σ **where** $\Sigma = (\lambda\sigma.\ \bigcup(\sigma 1, \sigma 2) \in pairs\ \sigma.\ add\text{-}states\ (upperize\ (\Sigma 1\ \sigma 1)\ (fvA\ Q1))\ (upperize\ (\Sigma 2\ \sigma 2)\ (fvA\ Q2)))$

show *?thesis*
proof (*rule hoare-triple-validI-bounded*)
show $\bigwedge s\ h\ n.\ (s, h), (s, h) \models Star\ P1\ P2 \Longrightarrow bounded\ h \Longrightarrow safe\ n\ \Delta\ (Cpar\ C1\ C2)\ (s, h)\ (\Sigma\ (s, h))$
proof –
fix *s h n* **assume** $(s, h), (s, h) \models Star\ P1\ P2$ *bounded h*
then obtain *h1 h2* **where** *asm0*: $Some\ h = Some\ h1 \oplus Some\ h2\ (s, h1), (s, h1) \models P1$
$(s, h2), (s, h2) \models P2$
using *always-sat-refl hyper-sat.simps*(*4*) **by** *blast*
then have $((s, h1), (s, h2)) \in pairs\ (s, h)$
using *pairs-def* **by** *blast*
then have $add\text{-}states\ (upperize\ (\Sigma 1\ (s, h1))\ (fvA\ Q1))\ (upperize\ (\Sigma 2\ (s, h2))\ (fvA\ Q2)) \subseteq \Sigma\ (s, h)$
using Σ-*def* **by** *blast*
moreover have $safe\ n\ \Delta\ (Cpar\ C1\ C2)\ (s, h)\ (add\text{-}states\ (upperize\ (\Sigma 1\ (s, h1))\ (fvA\ Q1))\ (upperize\ (\Sigma 2\ (s, h2))\ (fvA\ Q2)))$
proof (*rule parallel-comp*)
show $safe\ n\ \Delta\ C1\ (s, h1)\ (upperize\ (\Sigma 1\ (s, h1))\ (fvA\ Q1))$
by (*metis* ‹*bounded h*› *asm0*(*1*) *asm0*(*2*) *bounded-smaller-sum r1*(*1*) *safe-larger-set-aux snd-conv upperize-larger*)
show $safe\ n\ \Delta\ C2\ (s, h2)\ (upperize\ (\Sigma 2\ (s, h2))\ (fvA\ Q2))$
by (*metis* ‹*bounded h*› *asm0*(*1*) *asm0*(*3*) *bounded-smaller-sum plus-comm r2*(*1*) *safe-larger-set-aux snd-conv upperize-larger*)

show *Some h* = *Some h1* ⊕ *Some h2* **using** *asm0* **by** *simp*
show *disjoint* (*fvC C1* ∪ *fvA Q1*) (*wrC C2*)
by (*metis Un-subset-iff assms*(*3*) *disjoint-def inf-shunt*)
show *disjoint* (*fvC C2* ∪ *fvA Q2*) (*wrC C1*)
by (*metis Un-subset-iff assms*(*4*) *disjoint-def inf-shunt*)
show *upper-fvs* (*upperize* (Σ*1* (*s*, *h1*)) (*fvA Q1*)) (*fvA Q1*)
by (*simp add*: *upper-fvs-upperize*)
show *upper-fvs* (*upperize* (Σ*2* (*s*, *h2*)) (*fvA Q2*)) (*fvA Q2*)
using *upper-fvs-upperize* **by** *auto*
show ⋀Γ. Δ = *Some* Γ ⟹ *disjoint* (*fvA* (*invariant* Γ)) (*wrC C2*)
using *assms*(*5*) **by** *auto*
show ⋀Γ. Δ = *Some* Γ ⟹ *disjoint* (*fvA* (*invariant* Γ)) (*wrC C1*)
using *assms*(*6*) **by** *blast*
show *bounded h*
by (*simp add*: ‹*bounded h*›)
qed
ultimately show *safe n* Δ (*Cpar C1 C2*) (*s*, *h*) (Σ (*s*, *h*))
using *safe-larger-set* **by** *blast*
qed

fix *s h s′ h′*
assume (*s*, *h*), (*s′*, *h′*) ⊨ *Star P1 P2*
then obtain *h1 h2 h1′ h2′* **where** *asm0*: *Some h* = *Some h1* ⊕ *Some h2 Some h′* = *Some h1′* ⊕ *Some h2′*
(*s*, *h1*), (*s′*, *h1′*) ⊨ *P1* (*s*, *h2*), (*s′*, *h2′*) ⊨ *P2*
by *auto*

show *pair-sat* (Σ (*s*, *h*)) (Σ (*s′*, *h′*)) (*Star Q1 Q2*)
proof (*rule pair-satI*)
fix *ss hh ss′ hh′* **assume** *asm1*: (*ss*, *hh*) ∈ Σ (*s*, *h*) ∧ (*ss′*, *hh′*) ∈ Σ (*s′*, *h′*)

then obtain *σ1 σ2 σ1′ σ2′* **where** (*σ1*, *σ2*) ∈ *pairs* (*s*, *h*) (*σ1′*, *σ2′*) ∈ *pairs* (*s′*, *h′*)
(*ss*, *hh*) ∈ *add-states* (*upperize* (Σ*1 σ1*) (*fvA Q1*)) (*upperize* (Σ*2 σ2*) (*fvA Q2*))
(*ss′*, *hh′*) ∈ *add-states* (*upperize* (Σ*1 σ1′*) (*fvA Q1*)) (*upperize* (Σ*2 σ2′*) (*fvA Q2*))
using Σ-*def* **by** *blast*
then obtain *fst σ1* = *s fst σ2* = *s fst σ1′* = *s′ fst σ2′* = *s′ Some h* = *Some* (*snd σ1*) ⊕ *Some* (*snd σ2*)
Some h′ = *Some* (*snd σ1′*) ⊕ *Some* (*snd σ2′*)
(*s*, *snd σ1*), (*s*, *snd σ1*) ⊨ *P1* ∧ (*s*, *snd σ2*), (*s*, *snd σ2*) ⊨ *P2*
(*s′*, *snd σ1′*), (*s′*, *snd σ1′*) ⊨ *P1* ∧ (*s′*, *snd σ2′*), (*s′*, *snd σ2′*) ⊨ *P2*
using *case-prod-conv pairs-def* **by** *auto*

moreover have *snd σ1* = *h1* ∧ *snd σ2* = *h2* ∧ *snd σ1′* = *h1′* ∧ *snd σ2′* = *h2′*
proof (*cases precise P1*)
case *True*

then have *snd* $\sigma 1 = h1 \wedge$ *snd* $\sigma 1' = h1'$
proof (*rule preciseE*)
show $h \succeq h1 \wedge h \succeq$ *snd* $\sigma 1 \wedge h' \succeq h1' \wedge h' \succeq$ *snd* $\sigma 1'$
using *asm0*(*1*) *asm0*(*2*) *calculation*(*5*) *calculation*(*6*) *larger-def* **by** *blast*
show $(s, h1), (s', h1') \models P1 \wedge (s,$ *snd* $\sigma 1), (s',$ *snd* $\sigma 1') \models P1$
by (*metis True* ‹$h \succeq h1 \wedge h \succeq$ *snd* $\sigma 1 \wedge h' \succeq h1' \wedge h' \succeq$ *snd* $\sigma 1'$› *always-sat-refl asm0*(*3*) *calculation*(*7*) *calculation*(*8*) *preciseE sat-comm*)
qed
then show *?thesis*
by (*metis addition-cancellative asm0*(*1*) *asm0*(*2*) *calculation*(*5*) *calculation*(*6*) *plus-comm*)
next
case *False*
then have *precise P2*
using *assms*(*7*) **by** *blast*
then have *snd* $\sigma 2 = h2 \wedge$ *snd* $\sigma 2' = h2'$
proof (*rule preciseE*)
show $h \succeq h2 \wedge h \succeq$ *snd* $\sigma 2 \wedge h' \succeq h2' \wedge h' \succeq$ *snd* $\sigma 2'$
by (*metis asm0*(*1*) *asm0*(*2*) *calculation*(*5*) *calculation*(*6*) *larger-def plus-comm*)
show $(s, h2), (s', h2') \models P2 \wedge (s,$ *snd* $\sigma 2), (s',$ *snd* $\sigma 2') \models P2$
by (*metis* ‹$h \succeq h2 \wedge h \succeq$ *snd* $\sigma 2 \wedge h' \succeq h2' \wedge h' \succeq$ *snd* $\sigma 2'$› ‹*precise P2*› *always-sat-refl asm0*(*4*) *calculation*(*7*) *calculation*(*8*) *preciseE sat-comm*)
qed
then show *?thesis*
using *addition-cancellative asm0*(*1*) *asm0*(*2*) *calculation*(*5*) *calculation*(*6*) **by** *blast*
qed
ultimately have *pair-sat* $(\Sigma 1\ \sigma 1)$ $(\Sigma 1\ \sigma 1')$ *Q1* $\wedge$ *pair-sat* $(\Sigma 2\ \sigma 2)$ $(\Sigma 2\ \sigma 2')$ *Q2*
by (*metis asm0*(*3*) *asm0*(*4*) *prod.exhaust-sel r1*(*2*) *r2*(*2*))
then show $(ss, hh), (ss', hh') \models$ *Star Q1 Q2*
by (*metis* (*no-types*, *opaque-lifting*) ‹$(ss', hh') \in$ *add-states* (*upperize* $(\Sigma 1\ \sigma 1')$ (*fvA Q1*)) (*upperize* $(\Sigma 2\ \sigma 2')$ (*fvA Q2*))› ‹$(ss, hh) \in$ *add-states* (*upperize* $(\Sigma 1\ \sigma 1)$ (*fvA Q1*)) (*upperize* $(\Sigma 2\ \sigma 2)$ (*fvA Q2*))› *add-states-sat-star pair-sat-comm pair-sat-def pair-sat-upperize*)
qed
qed
qed

4.4.9 If

lemma *if-cases*:
assumes *red* (*Cif b C1 C2*) (s, h) C' (s', h')
and $C' = C1 \Longrightarrow s = s' \wedge h = h' \Longrightarrow$ *bdenot b s* $\Longrightarrow P$
and $C' = C2 \Longrightarrow s = s' \wedge h = h' \Longrightarrow \neg$ *bdenot b s* $\Longrightarrow P$
shows P
using *assms*(*1*)
apply (*rule red.cases*)

apply *blast+*
using *assms(2)* apply *fastforce*
using *assms(3)* apply *fastforce*
apply *blast+*
done

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

assumes *bdenot b s* $\Longrightarrow$ *safe n* Δ *C1* (*s*, *h*) *S*
and $\neg$ *bdenot b s* $\Longrightarrow$ *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 (*rule safeNoneI*)
show *Cif b C1 C2* = *Cskip* $\Longrightarrow$ (*s*, *h*) $\in$ *S* **by** *simp*
show *no-abort* (*None* :: (*'i*, *'a*, *nat*) *cont*) (*Cif b C1 C2*) *s h*
proof (*rule no-abortNoneI*)
fix *hf H* **assume** *Some H* = *Some h* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H*
show $\neg$ *aborts* (*Cif b C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*))
proof (*rule ccontr*)
assume $\neg$ $\neg$ *aborts* (*Cif b C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*))
then have *aborts* (*Cif b C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) **by** *simp*
then show *False*
by (*rule aborts.cases*) *auto*
qed
qed
fix *H hf C' s' h'*
assume *asm0*: *Some H* = *Some h* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *no-guard H*
$\wedge$ *red* (*Cif b C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C'* (*s'*, *h'*)
show $\exists$ *h'' H'*.
full-ownership (*get-fh H'*) $\wedge$
no-guard H' $\wedge$ *h'* = *FractionalHeap.normalize* (*get-fh H'*) $\wedge$ *Some H'* = *Some h''* $\oplus$ *Some hf* $\wedge$ *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C'* (*s'*, *h''*) *S*
by (*metis asm0 assms*(*1*) *assms*(*2*) *assms*(*3*) *if-cases*)
qed (*simp*)

lemma *if-safe-Some*:
assumes *bdenot b s* $\Longrightarrow$ *safe n* (*Some* Γ) *C1* (*s*, *h*) *S*
and $\neg$ *bdenot b s* $\Longrightarrow$ *safe n* (*Some* Γ) *C2* (*s*, *h*) *S*
shows *safe* (*Suc n*) (*Some* Γ) (*Cif b C1 C2*) (*s*, *h*) *S*
proof (*rule safeSomeI*)
show *Cif b C1 C2* = *Cskip* $\Longrightarrow$ (*s*, *h*) $\in$ *S* **by** *simp*
show *no-abort* (*Some* Γ) (*Cif b C1 C2*) *s h*
proof (*rule no-abortSomeI*)
fix *H hf hj v0*
assume *asm0*: *Some H* = *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$ *full-ownership*

(*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s hj* Γ
show $\neg$ *aborts* (*Cif b C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*))
proof (*rule ccontr*)
assume $\neg\neg$ *aborts* (*Cif b C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*))
then have *aborts* (*Cif b C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) **by** *simp*
then show *False*
by (*rule aborts.cases*) *auto*
qed
qed
fix *H hf C′ s′ h′ hj v0*
assume *asm0*: *Some H* = *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$ *full-ownership* (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H*
$\wedge$ *sat-inv s hj* Γ $\wedge$ *red* (*Cif b C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)
show $\exists$ *h′′ H′ hj′*.
full-ownership (*get-fh H′*) $\wedge$
semi-consistent Γ *v0 H′* $\wedge$
sat-inv s′ hj′ Γ $\wedge$ *h′* = *FractionalHeap.normalize* (*get-fh H′*) $\wedge$ *Some H′* = *Some h′′* $\oplus$ *Some hj′* $\oplus$ *Some hf* $\wedge$ *safe n* (*Some* Γ) *C′* (*s′*, *h′′*) *S*
by (*metis asm0 assms*(*1*) *assms*(*2*) *if-cases*)
qed (*simp*)

lemma *if-safe*:
fixes Δ :: (*′i*, *′a*, *nat*) *cont*
assumes *bdenot b s* $\Longrightarrow$ *safe n* Δ *C1* (*s*, *h*) *S*
and $\neg$ *bdenot b s* $\Longrightarrow$ *safe n* Δ *C2* (*s*, *h*) *S*
shows *safe* (*Suc n*) Δ (*Cif b C1 C2*) (*s*, *h*) *S*
apply (*cases* Δ)
using *assms*(*1*) *assms*(*2*) *if-safe-None* **apply** *blast*
using *assms*(*1*) *assms*(*2*) *if-safe-Some* **by** *blast*

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 –

obtain Σt **where** *safe-t*: $\bigwedge\sigma\ n.\ \sigma, \sigma \models$ *And P* (*Bool b*) $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe n* Δ *C1* σ ($\Sigma t\ \sigma$)
$\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models$ *And P* (*Bool b*) $\Longrightarrow$ *pair-sat* ($\Sigma t\ \sigma$) ($\Sigma t\ \sigma'$) *Q*
using *assms*(*1*) *hoare-triple-validE* **by** *blast*
obtain Σf **where** *safe-f*: $\bigwedge\sigma\ n.\ \sigma, \sigma \models$ *And P* (*Bool* (*Bnot b*)) $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe n* Δ *C2* σ ($\Sigma f\ \sigma$)
$\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models$ *And P* (*Bool* (*Bnot b*)) $\Longrightarrow$ *pair-sat* ($\Sigma f\ \sigma$) ($\Sigma f\ \sigma'$) *Q*
using *assms*(*2*) *hoare-triple-validE* **by** *blast*

define Σ **where** $\Sigma = (\lambda\sigma.\ \mathit{if}\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma)\ \mathit{then}\ \Sigma t\ \sigma\ \mathit{else}\ \Sigma f\ \sigma)$

show *?thesis*
proof (*rule hoare-triple-valid-smallerI-bounded*)

show $\bigwedge\sigma\ n.\ \sigma, \sigma \models \mathit{And}\ P\ (\mathit{Low}\ b) \Longrightarrow \mathit{bounded}\ (\mathit{snd}\ \sigma) \Longrightarrow \mathit{safe}\ n\ \Delta\ (\mathit{Cif}\ b\ C1\ C2)\ \sigma\ (\Sigma\ \sigma)$
proof –
fix σ n
assume *asm0*: $\sigma, \sigma \models \mathit{And}\ P\ (\mathit{Low}\ b)\ \mathit{bounded}\ (\mathit{snd}\ \sigma)$
show $\mathit{safe}\ n\ \Delta\ (\mathit{Cif}\ b\ C1\ C2)\ \sigma\ (\Sigma\ \sigma)$
proof (*cases bdenot b* (*fst* σ))
case *True*
then have $\mathit{safe}\ n\ \Delta\ C1\ \sigma\ (\Sigma\ \sigma)$
by (*metis* Σ*-def asm0 hyper-sat.simps*(*1*) *hyper-sat.simps*(*3*) *prod.exhaust-sel safe-t*(*1*))
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *Suc-n-not-le-n True if-safe nat-le-linear prod.exhaust-sel safe-smaller*)
next
case *False*
then have $\mathit{safe}\ n\ \Delta\ C2\ \sigma\ (\Sigma\ \sigma)$
by (*metis* Σ*-def asm0 bdenot.simps*(*3*) *hyper-sat.simps*(*1*) *hyper-sat.simps*(*3*) *prod.exhaust-sel safe-f*(*1*))
then show *?thesis*
by (*metis* (*mono-tags*) *False Suc-n-not-le-n if-safe nat-le-linear prod.exhaust-sel safe-smaller*)
qed
qed
fix σ σ' **assume** *asm0*: $\sigma, \sigma' \models \mathit{And}\ P\ (\mathit{Low}\ b)$
show $\mathit{pair\text{-}sat}\ (\Sigma\ \sigma)\ (\Sigma\ \sigma')\ Q$
proof (*cases bdenot b* (*fst* σ))
case *True*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) Σ*-def asm0 hyper-sat.simps*(*1*) *hyper-sat.simps*(*3*) *hyper-sat.simps*(*5*) *prod.exhaust-sel safe-t*(*2*))
next
case *False*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) Σ*-def asm0 bdenot.simps*(*3*) *hyper-sat.simps*(*1*) *hyper-sat.simps*(*3*) *hyper-sat.simps*(*5*) *prod.exhaust-sel safe-f*(*2*))
qed
qed
qed

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 –
obtain Σt **where** *safe-t*: $\bigwedge\sigma\ n.\ \sigma, \sigma \models And\ P\ (Bool\ b) \Longrightarrow bounded\ (snd\ \sigma) \Longrightarrow safe\ n\ \Delta\ C1\ \sigma\ (\Sigma t\ \sigma)$
$\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models And\ P\ (Bool\ b) \Longrightarrow pair\text{-}sat\ (\Sigma t\ \sigma)\ (\Sigma t\ \sigma')\ Q$
using *assms*(*1*) *hoare-triple-validE* **by** *blast*
obtain Σf **where** *safe-f*: $\bigwedge\sigma\ n.\ \sigma, \sigma \models And\ P\ (Bool\ (Bnot\ b)) \Longrightarrow bounded\ (snd\ \sigma) \Longrightarrow safe\ n\ \Delta\ C2\ \sigma\ (\Sigma f\ \sigma)$
$\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models And\ P\ (Bool\ (Bnot\ b)) \Longrightarrow pair\text{-}sat\ (\Sigma f\ \sigma)\ (\Sigma f\ \sigma')\ Q$
using *assms*(*2*) *hoare-triple-validE* **by** *blast*

define Σ **where** $\Sigma = (\lambda\sigma.\ if\ bdenot\ b\ (fst\ \sigma)\ then\ \Sigma t\ \sigma\ else\ \Sigma f\ \sigma)$

show *?thesis*
proof (*rule hoare-triple-valid-smallerI-bounded*)

show $\bigwedge\sigma\ n.\ \sigma, \sigma \models P \Longrightarrow bounded\ (snd\ \sigma) \Longrightarrow safe\ n\ \Delta\ (Cif\ b\ C1\ C2)\ \sigma\ (\Sigma\ \sigma)$
proof –
fix σ *n*
assume *asm0*: $\sigma, \sigma \models P$ *bounded* (*snd* σ)
show *safe n* Δ (*Cif b C1 C2*) σ (Σ σ)
proof (*cases bdenot b* (*fst* σ))
case *True*
then have *safe n* Δ *C1* σ (Σ σ)
by (*metis* Σ*-def asm0 hyper-sat.simps*(*1*) *hyper-sat.simps*(*3*) *prod.exhaust-sel safe-t*(*1*))
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *Suc-n-not-le-n True if-safe nat-le-linear prod.exhaust-sel safe-smaller*)
next
case *False*
then have *safe n* Δ *C2* σ (Σ σ)
by (*metis* Σ*-def asm0 bdenot.simps*(*3*) *hyper-sat.simps*(*1*) *hyper-sat.simps*(*3*) *prod.exhaust-sel safe-f*(*1*))
then show *?thesis*
by (*metis* (*mono-tags*) *False Suc-n-not-le-n if-safe nat-le-linear prod.exhaust-sel safe-smaller*)
qed
qed
fix $\sigma 1\ \sigma 2$ **assume** *asm0*: $\sigma 1, \sigma 2 \models P$
then have *asm0-bis*: $\sigma 2, \sigma 1 \models P$
by (*simp add*: *sat-comm*)
show *pair-sat* (Σ $\sigma 1$) (Σ $\sigma 2$) *Q*
proof (*rule pair-sat-smallerI*)
fix $\sigma 1'\ \sigma 2'$
assume *asm1*: $\sigma 1' \in \Sigma\ \sigma 1 \wedge \sigma 2' \in \Sigma\ \sigma 2$
then have $\sigma 1', \sigma 1' \models Q$

apply (*cases bdenot b* (*fst* $\sigma 1$))
apply (*metis* (*no-types*, *lifting*) Σ-*def always-sat-refl asm0 hyper-sat.simps*(*1*) *hyper-sat.simps*(*3*) *pair-sat-def safe-t*(*2*) *surjective-pairing*)
by (*metis* (*no-types*, *lifting*) Σ-*def always-sat-refl asm0 bdenot.simps*(*3*) *hyper-sat.simps*(*1*) *hyper-sat.simps*(*3*) *pair-satE prod.collapse safe-f*(*2*))
moreover have $\sigma 2'$, $\sigma 2' \models Q$
apply (*cases bdenot b* (*fst* $\sigma 2$))
apply (*metis* (*mono-tags*) Σ-*def always-sat-refl asm0-bis asm1 entailsI entails-def fst-conv hyper-sat.simps*(*1*) *hyper-sat.simps*(*3*) *old.prod.exhaust pair-sat-def safe-t*(*2*))
using Σ-*def always-sat-refl asm0-bis bdenot.simps*(*3*) *hyper-sat.simps*(*1*) *hyper-sat.simps*(*3*) *pair-satE prod.collapse safe-f*(*2*)
by (*metis* (*no-types*, *lifting*) *asm1*)
ultimately show $\sigma 1'$, $\sigma 2' \models Q$
by (*metis assms*(*3*) *eq-fst-iff unaryE*)
qed
qed
qed

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
using *aborts.simps assms cmd.inject*(*6*) **by** *blast*

lemma *safe-seq-None*:
assumes *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C1* (*s*, *h*) *S1*
and $\bigwedge m\ s'\ h'.\ m \le n \wedge (s', h') \in S1 \Longrightarrow$ *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*
using *assms*
proof (*induct n arbitrary*: *C1 s h*)
case (*Suc n*)
show *?case*
proof (*rule safeNoneI*)
show *no-abort* (*None* :: (*'i*, *'a*, *nat*) *cont*) (*Cseq C1 C2*) *s h*
by (*meson Suc.prems*(*1*) *aborts-seq-aborts-C1 no-abort.simps*(*1*) *safeNoneE-bis*(*2*))
show *accesses* (*Cseq C1 C2*) *s* $\subseteq$ *dom* (*fst h*) $\wedge$ *writes* (*Cseq C1 C2*) *s* $\subseteq$ *fpdom* (*fst h*)
by (*metis Suc.prems*(*1*) *accesses.simps*(*7*) *fst-conv safeAccessesE snd-conv writes.simps*(*7*))
fix *H hf* C' s' h'
assume *asm0*: *Some H* = *Some h* $\oplus$ *Some hf* $\wedge$
full-ownership (*get-fh H*) $\wedge$ *no-guard H* $\wedge$ *red* (*Cseq C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) C' (s', h')

show $\exists\, h''\ H'$.
full-ownership (*get-fh* H') $\wedge$
no-guard $H' \wedge h' =$ *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* $H' =$ *Some* $h'' \oplus$ *Some hf* $\wedge$ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (s', h'') *S2*
proof (*rule red-seq-cases*)
show *red* (*Cseq C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) C' (s', h')
using *asm0* **by** *blast*
show *C1* = *Cskip* $\Longrightarrow$
$C' = C2 \Longrightarrow$
(s', h') = (*s*, *FractionalHeap.normalize* (*get-fh H*)) $\Longrightarrow$
$\exists\, h''\ H'$. *full-ownership* (*get-fh* H') $\wedge$
no-guard $H' \wedge h' =$ *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* $H' =$ *Some* $h'' \oplus$ *Some hf* $\wedge$ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (s', h'') *S2*
using *Suc.prems*(*1*) *Suc.prems*(*2*) *asm0 order-refl prod.inject safeNoneE-bis*(*1*)
by (*metis le-SucI*)
fix *C1'* **assume** C' = *Cseq C1' C2 red C1* (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C1'* (s', h')
obtain H' h'' **where** *asm1*: *full-ownership* (*get-fh* H') *no-guard* H' $h' =$ *FractionalHeap.normalize* (*get-fh* H')
Some H' = *Some* $h'' \oplus$ *Some hf safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C1'* (s', h'') *S1*
using *Suc*(*2*) *safeNoneE*(*3*)[*of n C1 s h S1 H hf C1' s' h'*]
using ‹*red C1* (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C1'* (s', h')› *asm0* **by** *blast*
moreover have *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) (*Cseq C1' C2*) (s', h'') *S2*
using *Suc.hyps Suc.prems*(*2*) *calculation*(*5*)
using *le-Suc-eq* **by** *presburger*
ultimately show *?thesis*
using ‹C' = *Cseq C1' C2*› **by** *blast*
qed
qed (*simp*)
qed (*simp*)

lemma *safe-seq-Some*:
assumes *safe n* (*Some* Γ) *C1* (*s*, *h*) *S1*
and $\bigwedge m\ s'\ h'.\ m \le n \wedge (s', h') \in S1 \Longrightarrow$ *safe m* (*Some* Γ) *C2* (s', h') *S2*
shows *safe n* (*Some* Γ) (*Cseq C1 C2*) (*s*, *h*) *S2*
using *assms*
proof (*induct n arbitrary*: *C1 s h*)
case (*Suc n*)
show *?case*
proof (*rule safeSomeI*)
show *no-abort* (*Some* Γ) (*Cseq C1 C2*) *s h*
by (*meson Suc.prems*(*1*) *aborts-seq-aborts-C1 no-abort.simps*(*2*) *safeSomeE*(*2*))
show *accesses* (*Cseq C1 C2*) *s* ⊆ *dom* (*fst h*) ∧ *writes* (*Cseq C1 C2*) *s* ⊆ *fpdom* (*fst h*)
by (*metis Suc.prems*(*1*) *accesses.simps*(*7*) *fst-conv safeAccessesE snd-conv writes.simps*(*7*))
fix *H hf* C' s' h' *hj v0*

assume *asm0*: *Some H* = *Some h* ⊕ *Some hj* ⊕ *Some hf* ∧
full-ownership (*get-fh H*) ∧ *semi-consistent* Γ *v0 H* ∧ *sat-inv s hj* Γ ∧ *red* (*Cseq C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)
show ∃ *h″ H′ hj′*. *full-ownership* (*get-fh H′*) ∧
semi-consistent Γ *v0 H′* ∧ *sat-inv s′ hj′* Γ ∧ *h′* = *FractionalHeap.normalize* (*get-fh H′*) ∧ *Some H′* = *Some h″* ⊕ *Some hj′* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C′* (*s′*, *h″*) *S2*
proof (*rule red-seq-cases*)
show *red* (*Cseq C1 C2*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C′* (*s′*, *h′*)
using *asm0* **by** *blast*
show *C1* = *Cskip* ⟹
C′ = *C2* ⟹
(*s′*, *h′*) = (*s*, *FractionalHeap.normalize* (*get-fh H*)) ⟹ ∃ *h″ H′ hj′*. *full-ownership* (*get-fh H′*) ∧
semi-consistent Γ *v0 H′* ∧ *sat-inv s′ hj′* Γ ∧ *h′* = *FractionalHeap.normalize* (*get-fh H′*)
∧ *Some H′* = *Some h″* ⊕ *Some hj′* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C′* (*s′*, *h″*) *S2*
using *Pair-inject Suc.prems*(*1*) *Suc-n-not-le-n asm0 assms*(*2*) *not-less-eq-eq safeSomeE*(*1*)
by (*metis* (*no-types*, *lifting*) *Suc.prems*(*2*) *nat-le-linear*)
fix *C1′* **assume** *C′* = *Cseq C1′ C2 red C1* (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C1′* (*s′*, *h′*)
obtain *H′ h″ hj′* **where** *asm1*: *full-ownership* (*get-fh H′*) ∧
semi-consistent Γ *v0 H′* ∧ *sat-inv s′ hj′* Γ ∧ *h′* = *FractionalHeap.normalize* (*get-fh H′*) ∧ *Some H′* = *Some h″* ⊕ *Some hj′* ⊕ *Some hf* ∧ *safe n* (*Some* Γ) *C1′* (*s′*, *h″*) *S1*
using *Suc*(*2*) *safeSomeE*(*3*)[*of n* Γ *C1 s h S1 H hj hf v0 C1′ s′ h′*]
using ‹*red C1* (*s*, *FractionalHeap.normalize* (*get-fh H*)) *C1′* (*s′*, *h′*)› *asm0* **by** *blast*
moreover have *safe n* (*Some* Γ) (*Cseq C1′ C2*) (*s′*, *h″*) *S2*
by (*simp add*: *Suc.hyps Suc.prems*(*2*) *calculation*)
ultimately show *?thesis*
using ‹*C′* = *Cseq C1′ C2*› **by** *blast*
qed
qed (*simp*)
qed (*simp*)

lemma *seq-safe*:
fixes Δ :: (*′i*, *′a*, *nat*) *cont*
assumes *safe n* Δ *C1* (*s*, *h*) *S1*
and ⋀*m s′ h′*. *m* ≤ *n* ∧ (*s′*, *h′*) ∈ *S1* ⟹ *safe m* Δ *C2* (*s′*, *h′*) *S2*
shows *safe n* Δ (*Cseq C1 C2*) (*s*, *h*) *S2*
apply (*cases* Δ)
using *assms*(*1*) *assms*(*2*) *safe-seq-None* **apply** *blast*
using *assms*(*1*) *assms*(*2*) *safe-seq-Some* **by** *blast*

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 –
obtain $\Sigma 1$ **where** *safe-1*: $\bigwedge\sigma\ n.\ \sigma, \sigma \models P \Longrightarrow bounded\ (snd\ \sigma) \Longrightarrow safe\ n\ \Delta$ $C1\ \sigma\ (\Sigma 1\ \sigma)$
$\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models P \Longrightarrow pair\text{-}sat\ (\Sigma 1\ \sigma)\ (\Sigma 1\ \sigma')\ R$
using *assms*(*1*) *hoare-triple-validE* **by** *blast*
obtain $\Sigma 2$ **where** *safe-2*: $\bigwedge\sigma\ n.\ \sigma, \sigma \models R \Longrightarrow bounded\ (snd\ \sigma) \Longrightarrow safe\ n\ \Delta$ $C2\ \sigma\ (\Sigma 2\ \sigma)$
$\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models R \Longrightarrow pair\text{-}sat\ (\Sigma 2\ \sigma)\ (\Sigma 2\ \sigma')\ Q$
using *assms*(*2*) *hoare-triple-validE* **by** *blast*

define Σ **where** $\Sigma = (\lambda\sigma.\ (\bigcup\sigma' \in \Sigma 1\ \sigma.\ \Sigma 2\ \sigma'))$

show *?thesis*
proof (*rule hoare-triple-valid-smallerI-bounded*)
show $\bigwedge\sigma\ n.\ \sigma, \sigma \models P \Longrightarrow bounded\ (snd\ \sigma) \Longrightarrow safe\ n\ \Delta\ (Cseq\ C1\ C2)\ \sigma\ (\Sigma$ $\sigma)$
proof –
fix σ *n* **assume** *asm0*: $\sigma, \sigma \models P$ *bounded* (*snd* σ)
then have *pair-sat* ($\Sigma 1$ σ) ($\Sigma 1$ σ) *R*
using *safe-1*(*2*) **by** *blast*

have *safe n* Δ (*Cseq C1 C2*) (*fst* σ, *snd* σ) (Σ σ)
proof (*rule seq-safe*)
thm *restrict-safe-to-bounded*
show *safe n* Δ *C1* (*fst* σ, *snd* σ) (*Set.filter* (*bounded* $\circ$ *snd*) ($\Sigma 1$ σ))
apply (*rule restrict-safe-to-bounded*)
using *asm0 safe-1*(*1*) **by** *simp-all*

fix *m* s' h'
assume $m \leq n \wedge (s', h') \in Set.filter\ (bounded \circ snd)\ (\Sigma 1\ \sigma)$
then have *safe m* Δ *C2* (s', h') ($\Sigma 2$ (s', h'))
using *safe-2*(*1*)[*of* (s', h') *m*] ‹*pair-sat* ($\Sigma 1$ σ) ($\Sigma 1$ σ) *R*› **unfolding** *pair-sat-def*
by *simp*
then show *safe m* Δ *C2* (s', h') (Σ σ)
using *Sup-upper* ‹$\Sigma \equiv \lambda\sigma.\ \bigcup\ (\Sigma 2\ `\ \Sigma 1\ \sigma)$› ‹*pair-sat* ($\Sigma 1$ σ) ($\Sigma 1$ σ) *R*› *image-iff safe-larger-set asm0*(*2*)
by (*metis* (*no-types*, *lifting*) ‹$m \leq n \wedge (s', h') \in Set.filter\ (bounded \circ snd)$ $(\Sigma 1\ \sigma)$› *member-filter*)
qed
then show *safe n* Δ (*Cseq C1 C2*) σ (Σ σ) **by** *auto*
qed
fix $\sigma 1$ $\sigma 2$
assume *asm0*: $\sigma 1, \sigma 2 \models P$
show *pair-sat* (Σ $\sigma 1$) (Σ $\sigma 2$) *Q*
proof (*rule pair-sat-smallerI*)

fix $\sigma 1''$ $\sigma 2''$
assume *asm1*: $\sigma 1'' \in \Sigma\ \sigma 1 \wedge \sigma 2'' \in \Sigma\ \sigma 2$
then obtain $\sigma 1'$ $\sigma 2'$ **where** $\sigma 1'' \in \Sigma 2\ \sigma 1'$ $\sigma 1' \in \Sigma 1\ \sigma 1$ $\sigma 2'' \in \Sigma 2\ \sigma 2'$ $\sigma 2' \in \Sigma 1\ \sigma 2$
using ‹$\Sigma \equiv \lambda\sigma.\ \bigcup\ (\Sigma 2$ ‘ $\Sigma 1\ \sigma)$› **by** *blast*
then show $\sigma 1''$, $\sigma 2'' \models Q$
by (*meson asm0 pair-sat-def safe-1*(*2*) *safe-2*(*2*))
qed
qed
qed

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* ⊕ *Some hf0*
and *bounded H*
shows *safe n* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C* (*s*, *H*) (*add-states S* {(*s''*, *hf0*) |*s''*. *agrees* (− *wrC C*) *s s''*})
using *assms*
proof (*induct n arbitrary*: *s h H C*)
case (*Suc n*)
show *safe* (*Suc n*) (*None* :: (*'i*, *'a*, *nat*) *cont*) *C* (*s*, *H*) (*add-states S* {(*s''*, *hf0*) |*s''*. *agrees* (− *wrC C*) *s s''*})
proof (*rule safeNoneI*)
show *C* = *Cskip* ⟹ (*s*, *H*) ∈ *add-states S* {(*s'*, *hf0*) |*s'*. *agrees* (− *wrC C*) *s s'*}
using *CollectI Suc.prems*(*1*) *Suc.prems*(*2*) *add-states-def agrees-def*[*of* − *wrC C s*] *safeNoneE*(*1*)[*of n C s h S*]
by *fast*
show *no-abort* (*None* :: (*'i*, *'a*, *nat*) *cont*) *C s H*
using *Suc.prems*(*1*) *Suc.prems*(*2*) *larger-def no-abort-larger safeNoneE*(*2*) **by** *blast*

have *accesses C s* ⊆ *dom* (*fst h*) ∧ *writes C s* ⊆ *fpdom* (*fst h*)
using *Suc.prems*(*1*) **by** *auto*
moreover have *dom* (*fst h*) ⊆ *dom* (*fst H*)
by (*metis Suc.prems*(*2*) *addition-smaller-domain get-fh.simps*)
moreover have *fpdom* (*fst h*) ⊆ *fpdom* (*fst H*)
using *Suc.prems*(*2*) *fpdom-inclusion*
using *Suc.prems*(*3*) **by** *blast*

ultimately show *accesses C s* ⊆ *dom* (*fst H*) ∧ *writes C s* ⊆ *fpdom* (*fst H*)
by *blast*

fix *H1 hf1 C' s' h'*
assume *asm0*: *Some H1* = *Some H* ⊕ *Some hf1* ∧ *full-ownership* (*get-fh H1*) ∧ *no-guard H1* ∧ *red C* (*s*, *FractionalHeap.normalize* (*get-fh H1*)) *C'* (*s'*, *h'*)
then obtain *hf* **where** *Some hf* = *Some hf0* ⊕ *Some hf1*

by (*metis* (*no-types*, *opaque-lifting*) *Suc.prems*(*2*) *option.collapse plus.simps*(*1*) *plus-asso plus-comm*)
then have *Some H1* = *Some h* ⊕ *Some hf*
by (*metis Suc.prems*(*2*) *asm0 plus-asso*)
then obtain h'' H' **where** *r*: *full-ownership* (*get-fh* H')
no-guard H' h' = *FractionalHeap.normalize* (*get-fh* H') *Some* H' = *Some* h'' ⊕ *Some hf safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (s', h'') *S*
using *safeNoneE*(*3*)[*of n C s h S H1 hf* C' s'] *Suc.prems*(*1*) *asm0* **by** *blast*
then obtain h''' **where** *Some* h''' = *Some* h'' ⊕ *Some hf0*
by (*metis* ‹*Some hf* = *Some hf0* ⊕ *Some hf1*› *not-Some-eq plus.simps*(*1*) *plus-asso*)
then have *Some* H' = *Some* h''' ⊕ *Some hf1*
by (*metis* ‹*Some hf* = *Some hf0* ⊕ *Some hf1*› *plus-asso r*(*4*))
moreover have *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (s', h''') (*add-states S* {(s'', *hf0*) |s''. *agrees* (− *wrC* C') s' s''})
proof (*rule Suc.hyps*)
show *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (s', h'') *S*
using *r* **by** *simp*
show *Some* h''' = *Some* h'' ⊕ *Some hf0*
by (*simp add*: ‹*Some* h''' = *Some* h'' ⊕ *Some hf0*›)
show *bounded* h'''
by (*metis bounded-smaller-sum calculation full-ownership-then-bounded get-fh.simps r*(*1*))
qed
moreover have *add-states S* {(s'', *hf0*) |s''. *agrees* (− *wrC* C') s' s''} ⊆ *add-states S* {(s'', *hf0*) |s''. *agrees* (− *wrC C*) *s* s''}
proof −
have *wrC* C' ⊆ *wrC C*
using *asm0 red-properties*(*1*) **by** *blast*
have {(s'', *hf0*) |s''. *agrees* (− *wrC* C') s' s''} ⊆ {(s'', *hf0*) |s''. *agrees* (− *wrC C*) *s* s''}
proof
fix *x* **assume** *x* ∈ {(s'', *hf0*) |s''. *agrees* (− *wrC* C') s' s''}
then have *agrees* (− *wrC* C') s' (*fst x*) ∧ *snd x* = *hf0* **by** *force*
moreover have *fvC* C' ⊆ *fvC C* ∧ *wrC* C' ⊆ *wrC C* ∧ *agrees* (− *wrC C*) s' *s*
using *asm0 red-properties*(*1*) **by** *force*

moreover have *agrees* (− *wrC C*) *s* (*fst x*)
proof (*rule agreesI*)
fix *y* **assume** *y* ∈ − *wrC C*
show *s y* = *fst x y*
by (*metis* (*no-types*, *lifting*) *Compl-subset-Compl-iff* ‹*y* ∈ − *wrC C*› *agrees-def calculation*(*1*) *calculation*(*2*) *in-mono*)
qed
then show *x* ∈ {(s'', *hf0*) |s''. *agrees* (− *wrC C*) *s* s''}
using ‹*agrees* (− *wrC* C') s' (*fst x*) ∧ *snd x* = *hf0*› **by** *force*
qed
then show *?thesis*

by (*metis* (*no-types*, *lifting*) *add-states-comm add-states-subset*)
qed
ultimately have *safe n* (*None* :: ($'i$, $'a$, *nat*) *cont*) C' (s', h''') (*add-states S* $\{(s'', hf0) \mid s''.$ *agrees* ($-$ *wrC C*) *s* $s''\}$)
using *safe-larger-set* **by** *blast*
then show $\exists h''\ H'$.
full-ownership (*get-fh* H') $\wedge$
no-guard H' $\wedge$
h' = *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* H' = *Some* h'' $\oplus$ *Some hf1* $\wedge$ *safe n* (*None* :: ($'i$, $'a$, *nat*) *cont*) C' (s', h'') (*add-states S* $\{(s'', hf0) \mid s''.$ *agrees* ($-$ *wrC C*) *s* $s''\}$)
using ‹*Some* H' = *Some* h''' $\oplus$ *Some hf1*› *r*(*1*) *r*(*2*) *r*(*3*) **by** *blast*
qed
qed (*simp*)

lemma *safe-frame-Some*:
assumes *safe n* (*Some* Γ) *C* (*s*, *h*) *S*
and *Some H* = *Some h* $\oplus$ *Some hf0*
and *bounded H*
shows *safe n* (*Some* Γ) *C* (*s*, *H*) (*add-states S* $\{(s'', hf0) \mid s''.$ *agrees* ($-$ *wrC C*) *s* $s''\}$)
using *assms*
proof (*induct n arbitrary*: *s h H C*)
case (*Suc n*)
let *?R* = $\{(s'', hf0) \mid s''.$ *agrees* ($-$ *wrC C*) *s* $s''\}$
show *safe* (*Suc n*) (*Some* Γ) *C* (*s*, *H*) (*add-states S ?R*)
proof (*rule safeSomeI*)
show *C* = *Cskip* $\Longrightarrow$ (*s*, *H*) $\in$ *add-states S ?R*
using *CollectI Suc.prems*(*1*) *Suc.prems*(*2*) *add-states-def*[*of S ?R*] *agrees-def*[*of* $-$ *wrC C s*]
safeSomeE(*1*)[*of n* Γ *C s h S*] **by** *fast*
show *no-abort* (*Some* Γ) *C s H*
using *Suc.prems*(*1*) *Suc.prems*(*2*) *larger-def no-abort-larger safeSomeE*(*2*) **by** *blast*

have *accesses C s* $\subseteq$ *dom* (*fst h*) $\wedge$ *writes C s* $\subseteq$ *fpdom* (*fst h*)
using *Suc.prems*(*1*) **by** *auto*
moreover have *dom* (*fst h*) $\subseteq$ *dom* (*fst H*)
by (*metis Suc.prems*(*2*) *addition-smaller-domain get-fh.simps*)
moreover have *fpdom* (*fst h*) $\subseteq$ *fpdom* (*fst H*)
using *Suc.prems*(*2*) *Suc.prems*(*3*) *fpdom-inclusion* **by** *blast*

ultimately show *accesses C s* $\subseteq$ *dom* (*fst H*) $\wedge$ *writes C s* $\subseteq$ *fpdom* (*fst H*)
by *blast*

fix *H1 hf1* C' s' h' *hj v0*
assume *asm0*: *Some H1* = *Some H* $\oplus$ *Some hj* $\oplus$ *Some hf1* $\wedge$
full-ownership (*get-fh H1*) $\wedge$ *semi-consistent* Γ *v0 H1* $\wedge$ *sat-inv s hj* Γ $\wedge$ *red C* (*s*, *FractionalHeap.normalize* (*get-fh H1*)) C' (s', h')

then obtain *hf* **where** *Some hf = Some hf0* ⊕ *Some hf1*
by (*metis* (*no-types*, *opaque-lifting*) *Suc.prems*(*2*) *option.collapse plus.simps*(*1*) *plus-asso plus-comm*)
then have *Some H1 = Some h* ⊕ *Some hj* ⊕ *Some hf*
by (*metis* (*no-types*, *opaque-lifting*) *Suc.prems*(*2*) *asm0 plus-asso plus-comm*)

then obtain h'' H' hj' **where** *r*: *full-ownership* (*get-fh* H') ∧ *semi-consistent* Γ *v0* H' ∧ *sat-inv* s' hj' Γ ∧ h' = *FractionalHeap.normalize* (*get-fh* H') ∧ *Some* H' = *Some* h'' ⊕ *Some* hj' ⊕ *Some hf* ∧ *safe n* (*Some* Γ) C' (s', h'') *S*
using *safeSomeE*(*3*)[*of n* Γ *C s h S H1 hj hf v0* C' s' h'] *Suc.prems*(*1*) *asm0* **by** *blast*

then obtain h''' **where** *Some* h''' = *Some* h'' ⊕ *Some hf0*
by (*metis* (*no-types*, *lifting*) ‹*Some hf = Some hf0* ⊕ *Some hf1*› *plus.simps*(*2*) *plus.simps*(*3*) *plus-asso plus-comm*)
then have *Some* H' = *Some* h''' ⊕ *Some* hj' ⊕ *Some hf1*
by (*metis* (*no-types*, *lifting*) ‹*Some hf = Some hf0* ⊕ *Some hf1*› *plus-asso plus-comm r*)
moreover have *safe n* (*Some* Γ) C' (s', h''') (*add-states S* {(s'', *hf0*) |s''. *agrees* (− *wrC* C') s' s''})
proof (*rule Suc.hyps*)
show *safe n* (*Some* Γ) C' (s', h'') *S*
using *r* **by** *simp*
show *Some* h''' = *Some* h'' ⊕ *Some hf0*
by (*simp add*: ‹*Some* h''' = *Some* h'' ⊕ *Some hf0*›)
show *bounded* h'''
apply (*rule bounded-smaller*[*of* H'])
apply (*metis full-ownership-then-bounded get-fh.simps r*)
by (*simp add*: *calculation larger3 plus-comm*)
qed
moreover have *add-states S* {(s'', *hf0*) |s''. *agrees* (− *wrC* C') s' s''} ⊆ *add-states S* {(s'', *hf0*) |s''. *agrees* (− *wrC C*) *s* s''}
proof −
have *wrC* C' ⊆ *wrC C*
using *asm0 red-properties*(*1*) **by** *blast*
have {(s'', *hf0*) |s''. *agrees* (− *wrC* C') s' s''} ⊆ {(s'', *hf0*) |s''. *agrees* (− *wrC C*) *s* s''}
proof
fix *x* **assume** *x* ∈ {(s'', *hf0*) |s''. *agrees* (− *wrC* C') s' s''}
then have *agrees* (− *wrC* C') s' (*fst x*) ∧ *snd x = hf0* **by** *force*
moreover have *fvC* C' ⊆ *fvC C* ∧ *wrC* C' ⊆ *wrC C* ∧ *agrees* (− *wrC C*) s' *s*
using *asm0 red-properties*(*1*) **by** *force*
moreover have *agrees* (− *wrC C*) *s* (*fst x*)
proof (*rule agreesI*)
fix *y* **assume** *y* ∈ − *wrC C*
then show *s y = fst x y*

by (*metis* (*mono-tags*, *opaque-lifting*) *Compl-iff agrees-def calculation*(*1*) *calculation*(*2*) *in-mono*)
qed
then show $x \in \{(s'', hf0) \mid s''.\ agrees\ (-\ wrC\ C)\ s\ s''\}$
using ‹*agrees* (− *wrC C*′) *s*′ (*fst x*) ∧ *snd x* = *hf0*› **by** *force*
qed
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *add-states-comm add-states-subset*)
qed
ultimately have *safe n* (*Some* Γ) *C*′ (*s*′, *h*′′′) (*add-states S ?R*)
using *safe-larger-set* **by** *blast*
then show ∃ *h*′′ *H*′ *hj*′. *full-ownership* (*get-fh H*′) ∧
semi-consistent Γ *v0 H*′ ∧
sat-inv s′ *hj*′ Γ ∧ *h*′ = *FractionalHeap.normalize* (*get-fh H*′) ∧ *Some H*′ = *Some h*′′ ⊕ *Some hj*′ ⊕ *Some hf1* ∧ *safe n* (*Some* Γ) *C*′ (*s*′, *h*′′) (*add-states S ?R*)
using ‹*Some H*′ = *Some h*′′′ ⊕ *Some hj*′ ⊕ *Some hf1*› *r* **by** *blast*
qed
qed (*simp*)

lemma *safe-frame*:
fixes Δ :: (′*i*, ′*a*, *nat*) *cont*
assumes *safe n* Δ *C* (*s*, *h*) *S*
and *Some H* = *Some h* ⊕ *Some hf0*
and *bounded H*
shows *safe n* Δ *C* (*s*, *H*) (*add-states S* {(*s*′′, *hf0*) |*s*′′. *agrees* (− *wrC C*) *s s*′′})
apply (*cases* Δ)
using *assms safe-frame-None* **apply** *blast*
using *assms safe-frame-Some* **by** *blast*

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 −
obtain Σ **where** *asm0*: ⋀σ *n*. σ, σ ⊨ *P* ⟹ *bounded* (*snd* σ) ⟹ *safe n* Δ *C* σ (Σ σ) ⋀σ σ′. σ, σ′ ⊨ *P* ⟹ *pair-sat* (Σ σ) (Σ σ′) *Q*
using *assms*(*1*) *hoare-triple-validE* **by** *blast*

define *pairs* **where** *pairs* = (λσ. { (*p*, *r*) |*p r*. *Some* (*snd* σ) = *Some p* ⊕ *Some r* ∧ (*fst* σ, *p*), (*fst* σ, *p*) ⊨ *P*
∧ (*fst* σ, *r*), (*fst* σ, *r*) ⊨ *R* })

define Σ′ **where** Σ′ = (λσ. (⋃(*p*, *r*) ∈ *pairs* σ. *add-states* (Σ (*fst* σ, *p*)) {(*s*′′, *r*) |*s*′′. *agrees* (− *wrC C*) (*fst* σ) *s*′′}))

show *?thesis*
proof (*rule hoare-triple-validI-bounded*)

show $\bigwedge s\ h\ n.\ (s,\ h),\ (s,\ h) \models Star\ P\ R \Longrightarrow bounded\ h \Longrightarrow safe\ n\ \Delta\ C\ (s,\ h)\ (\Sigma'\ (s,\ h))$
proof −
fix *s h n* **assume** *asm1*: $(s,\ h),\ (s,\ h) \models Star\ P\ R\ bounded\ h$
then obtain *p r* **where** $Some\ h = Some\ p \oplus Some\ r\ (s,\ p),\ (s,\ p) \models P\ (s,\ r),\ (s,\ r) \models R$
using *always-sat-refl hyper-sat.simps(4)* **by** *blast*
then have $safe\ n\ \Delta\ C\ (s,\ p)\ (\Sigma\ (s,\ p))$
using *asm0 asm1*
by (*metis bounded-smaller-sum snd-conv*)

then have $safe\ n\ \Delta\ C\ (s,\ h)\ (add\text{-}states\ (\Sigma\ (s,\ p))\ \{(s'',\ r)\ |s''.\ agrees\ (-\ wrC\ C)\ s\ s''\})$
using *safe-frame*[*of* $n\ \Delta\ C\ s\ p\ \Sigma\ (s,\ p)\ h\ r$] ‹$Some\ h = Some\ p \oplus Some\ r$›
using *asm1(2)* **by** *blast*
moreover have $(add\text{-}states\ (\Sigma\ (s,\ p))\ \{(s'',\ r)\ |s''.\ agrees\ (-\ wrC\ C)\ s\ s''\}) \subseteq \Sigma'\ (s,\ h)$
proof −
have $(p,\ r) \in pairs\ (s,\ h)$
using ‹$(s,\ p),\ (s,\ p) \models P$› ‹$(s,\ r),\ (s,\ r) \models R$› ‹$Some\ h = Some\ p \oplus Some\ r$› *pairs-def* **by** *force*
then show *?thesis*
using Σ'*-def* **by** *auto*
qed
ultimately show $safe\ n\ \Delta\ C\ (s,\ h)\ (\Sigma'\ (s,\ h))$
using *safe-larger-set* **by** *blast*
qed

fix *s1 h1 s2 h2*
assume *asm1*: $(s1,\ h1),\ (s2,\ h2) \models Star\ P\ R$
then obtain *p1 p2 r1 r2* **where** $Some\ h1 = Some\ p1 \oplus Some\ r1\ Some\ h2 = Some\ p2 \oplus Some\ r2$
$(s1,\ p1),\ (s2,\ p2) \models P\ (s1,\ r1),\ (s2,\ r2) \models R$
by *auto*
then have $(s1,\ p1),\ (s1,\ p1) \models P \wedge (s1,\ r1),\ (s1,\ r1) \models R \wedge (s2,\ p2),\ (s2,\ p2) \models P \wedge (s2,\ r2),\ (s2,\ r2) \models R$
using *always-sat-refl sat-comm* **by** *blast*

show $pair\text{-}sat\ (\Sigma'\ (s1,\ h1))\ (\Sigma'\ (s2,\ h2))\ (Star\ Q\ R)$
proof (*rule pair-satI*)
fix *s1' h1' s2' h2'*
assume *asm2*: $(s1',\ h1') \in \Sigma'\ (s1,\ h1) \wedge (s2',\ h2') \in \Sigma'\ (s2,\ h2)$
then obtain *p1' r1' p2' r2'* **where** $(p1',\ r1') \in pairs\ (s1,\ h1)\ (p2',\ r2') \in pairs\ (s2,\ h2)$
$(s1',\ h1') \in add\text{-}states\ (\Sigma\ (s1,\ p1'))\ \{(s'',\ r1')\ |s''.\ agrees\ (-\ wrC\ C)\ s1\ s''\}$
$(s2',\ h2') \in add\text{-}states\ (\Sigma\ (s2,\ p2'))\ \{(s'',\ r2')\ |s''.\ agrees\ (-\ wrC\ C)\ s2\ s''\}$

using Σ'*-def* **by** *force*

moreover obtain $(s1,\ p1'),\ (s1,\ p1') \models P\ (s1,\ r1'),\ (s1,\ r1') \models R\ (s2,\ p2'),\ (s2,\ p2') \models P\ (s2,\ r2'),\ (s2,\ r2') \models R$
Some h1 = *Some p1'* ⊕ *Some r1' Some h2* = *Some p2'* ⊕ *Some r2'*
using *calculation*(*1*) *calculation*(*2*) *pairs-def* **by** *auto*
ultimately have $p1 = p1' \wedge p2 = p2' \wedge r1 = r1' \wedge r2 = r2'$
proof (*cases precise P*)
case *True*
then have $p1 = p1' \wedge p2 = p2'$ **using** *preciseE*
by (*metis* ‹$(s1,\ p1),\ (s1,\ p1) \models P \wedge (s1,\ r1),\ (s1,\ r1) \models R \wedge (s2,\ p2),\ (s2,\ p2) \models P \wedge (s2,\ r2),\ (s2,\ r2) \models R$› ‹*Some h1* = *Some p1* ⊕ *Some r1*› ‹*Some h2* = *Some p2* ⊕ *Some r2*› ‹$\bigwedge$*thesis*. (⟦$(s1,\ p1'),\ (s1,\ p1') \models P$; $(s1,\ r1'),\ (s1,\ r1') \models R$; $(s2,\ p2'),\ (s2,\ p2') \models P$; $(s2,\ r2'),\ (s2,\ r2') \models R$; *Some h1* = *Some p1'* ⊕ *Some r1'*; *Some h2* = *Some p2'* ⊕ *Some r2'*⟧ $\Longrightarrow$ *thesis*) $\Longrightarrow$ *thesis*› *larger-def*)
then show *?thesis*
by (*metis* ‹*Some h1* = *Some p1* ⊕ *Some r1*› ‹*Some h1* = *Some p1'* ⊕ *Some r1'*› ‹*Some h2* = *Some p2* ⊕ *Some r2*› ‹*Some h2* = *Some p2'* ⊕ *Some r2'*› *addition-cancellative plus-comm*)
next
case *False*
then have *precise R*
using *assms*(*3*) **by** *auto*
then show *?thesis*
by (*metis* (*no-types*, *opaque-lifting*) ‹$(s1,\ p1),\ (s1,\ p1) \models P \wedge (s1,\ r1),\ (s1,\ r1) \models R \wedge (s2,\ p2),\ (s2,\ p2) \models P \wedge (s2,\ r2),\ (s2,\ r2) \models R$› ‹*Some h1* = *Some p1* ⊕ *Some r1*› ‹*Some h2* = *Some p2* ⊕ *Some r2*› ‹$\bigwedge$*thesis*. (⟦$(s1,\ p1'),\ (s1,\ p1') \models P$; $(s1,\ r1'),\ (s1,\ r1') \models R$; $(s2,\ p2'),\ (s2,\ p2') \models P$; $(s2,\ r2'),\ (s2,\ r2') \models R$; *Some h1* = *Some p1'* ⊕ *Some r1'*; *Some h2* = *Some p2'* ⊕ *Some r2'*⟧ $\Longrightarrow$ *thesis*) $\Longrightarrow$ *thesis*› *addition-cancellative larger-def plus-comm preciseE*)
qed
then have *pair-sat* $(\Sigma\ (s1,\ p1'))\ (\Sigma\ (s2,\ p2'))\ Q$
using ‹$(s1,\ p1),\ (s2,\ p2) \models P$› *asm0*(*2*) **by** *blast*
moreover have *pair-sat* $\{(s'',\ r1')\ |s''.\ agrees\ (-\ wrC\ C)\ s1\ s''\}\ \{(s'',\ r2')\ |s''.\ agrees\ (-\ wrC\ C)\ s2\ s''\}\ R$
(**is** *pair-sat ?R1 ?R2 R*)
proof (*rule pair-satI*)
fix *s1'' r1'' s2'' r2''* **assume** $(s1'',\ r1'') \in \{(s'',\ r1')\ |s''.\ agrees\ (-\ wrC\ C)\ s1\ s''\} \wedge (s2'',\ r2'') \in \{(s'',\ r2')\ |s''.\ agrees\ (-\ wrC\ C)\ s2\ s''\}$
then obtain $r1'' = r1'\ r2'' = r2'\ agrees\ (-\ wrC\ C)\ s1\ s1''\ agrees\ (-\ wrC\ C)\ s2\ s2''$
by *fastforce*
then show $(s1'',\ r1''),\ (s2'',\ r2'') \models R$
using ‹$(s1,\ r1),\ (s2,\ r2) \models R$› ‹$p1 = p1' \wedge p2 = p2' \wedge r1 = r1' \wedge r2 = r2'$› *agrees-minusD agrees-same*
assms(*2*) *sat-comm*
by (*metis* (*no-types*, *opaque-lifting*) *disjoint-def inf-commute*)
qed
ultimately have *pair-sat* (*add-states* $(\Sigma\ (s1,\ p1'))$ *?R1*) (*add-states* (Σ (*s2*,

p2′)) *?R2*) (*Star Q R*)
        using *add-states-sat-star* **by** *blast*
      then show (*s1′*, *h1′*), (*s2′*, *h2′*) $\models$ *Star Q R*
        using ‹(*s1′*, *h1′*) $\in$ *add-states* (Σ (*s1*, *p1′*)) {(*s″*, *r1′*) |*s″*. *agrees* ($-$ *wrC C*) *s1 s″*}› ‹(*s2′*, *h2′*) $\in$ *add-states* (Σ (*s2*, *p2′*)) {(*s″*, *r2′*) |*s″*. *agrees* ($-$ *wrC C*) *s2 s″*}› *pair-sat-def* **by** *blast*
    qed
  qed
qed

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 $-$
  obtain Σ **where** *asm0*: $\bigwedge\sigma\ n.\ \sigma, \sigma \models P' \Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe* $n\ \Delta\ C\ \sigma$ ($\Sigma\ \sigma$) $\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models P' \Longrightarrow$ *pair-sat* ($\Sigma\ \sigma$) ($\Sigma\ \sigma'$) Q'
    using *assms*(*1*) *hoare-triple-validE* **by** *blast*

  show *?thesis*
  proof (*rule hoare-triple-validI-bounded*)
    show $\bigwedge s\ h\ n.\ (s, h), (s, h) \models P \Longrightarrow$ *bounded* $h \Longrightarrow$ *safe* $n\ \Delta\ C\ (s, h)\ (\Sigma\ (s, h))$
      using *asm0*(*1*) *assms*(*2*) *entails-def*
      by *fastforce*
    show $\bigwedge s\ h\ s'\ h'.\ (s, h), (s', h') \models P \Longrightarrow$ *pair-sat* (Σ (s, h)) (Σ (s', h')) Q
      by (*meson asm0*(*2*) *assms*(*2*) *assms*(*3*) *entails-def pair-sat-def*)
  qed
qed

4.4.13 Existential

theorem *existential-rule*:
  fixes Δ :: (*′i*, *′a*, *nat*) *cont*
  assumes *hoare-triple-valid* Δ P C Q
      and $x \notin$ *fvC* C
      and $\bigwedge\Gamma.\ \Delta =$ *Some* $\Gamma \Longrightarrow x \notin$ *fvA* (*invariant* Γ)
      and *unambiguous* P x
    shows *hoare-triple-valid* Δ (*Exists* x P) C (*Exists* x Q)
proof $-$
  obtain Σ **where** *asm0*: $\bigwedge\sigma\ n.\ \sigma, \sigma \models P \Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe* $n\ \Delta\ C\ \sigma$ ($\Sigma\ \sigma$) $\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models P \Longrightarrow$ *pair-sat* ($\Sigma\ \sigma$) ($\Sigma\ \sigma'$) Q
    using *assms*(*1*) *hoare-triple-validE* **by** *blast*

  define Σ' **where** $\Sigma' = (\lambda\sigma.\ \bigcup v \in \{\ v\ |v.\ ((fst\ \sigma)(x := v),\ snd\ \sigma), ((fst\ \sigma)(x := v),\ snd\ \sigma) \models P\ \}.\ upperize\ (\Sigma\ ((fst\ \sigma)(x := v),\ snd\ \sigma))\ (fvA\ Q - \{x\}))$

```
  show ?thesis
  proof (rule hoare-triple-validI-bounded)
    show ⋀s h n. (s, h), (s, h) ⊨ Exists x P ⟹ bounded h ⟹ safe n Δ C (s, h)
(Σ' (s, h))
    proof -
      fix s h n assume (s, h), (s, h) ⊨ Exists x P bounded h
      then obtain v where (s(x := v), h), (s(x := v), h) ⊨ P
        using always-sat-refl hyper-sat.simps(7) by blast
      then have Σ (s(x := v), h) ⊆ Σ' (s, h)
        using upperize-larger SUP-upper2 Σ'-def by fastforce

      moreover have safe n Δ C (s(x := v), h) (Σ (s(x := v), h))
        by (simp add: ‹(s(x := v), h), (s(x := v), h) ⊨ P› ‹bounded h› asm0(1))
      ultimately have safe n Δ C (s(x := v), h) (Σ' (s, h))
        using safe-larger-set by blast
      then have safe n Δ C (s, h) (Σ' (s, h))
      proof (rule safe-free-vars)
        show ⋀Γ. Δ = Some Γ ⟹ agrees (fvA (invariant Γ)) (s(x := v)) s
          by (meson agrees-comm agrees-update assms(3))
        show agrees (fvC C ∪ (fvA Q − {x})) (s(x := v)) s
          by (simp add: agrees-def assms(2))
        show upper-fvs (Σ' (s, h)) (fvA Q − {x})
        proof (rule upper-fvsI)
          fix sa s' ha
          assume asm0: (sa, ha) ∈ Σ' (s, h) ∧ agrees (fvA Q − {x}) sa s'
            then obtain v where (s(x := v), h), (s(x := v), h) ⊨ P (sa, ha) ∈
upperize (Σ (s(x := v), h)) (fvA Q − {x})
            using Σ'-def by force
          then have (s', ha) ∈ upperize (Σ (s(x := v), h)) (fvA Q − {x})
            using asm0 upper-fvs-def upper-fvs-upperize by blast
          then show (s', ha) ∈ Σ' (s, h)
            using ‹(s(x := v), h), (s(x := v), h) ⊨ P› Σ'-def by force
        qed
      qed
      then show safe n Δ C (s, h) (Σ' (s, h))
        by auto
    qed
    fix s1 h1 s2 h2
    assume asm1: (s1, h1), (s2, h2) ⊨ Exists x P
     then obtain v1' v2' where (s1(x := v1'), h1), (s2(x := v2'), h2) ⊨ P by
auto
    show pair-sat (Σ' (s1, h1)) (Σ' (s2, h2)) (Exists x Q)
    proof (rule pair-satI)
      fix s1' h1' s2' h2'
      assume asm2: (s1', h1') ∈ Σ' (s1, h1) ∧ (s2', h2') ∈ Σ' (s2, h2)

      then obtain v1 v2 where
        r: (s1(x := v1), h1), (s1(x := v1), h1) ⊨ P (s1', h1') ∈ upperize (Σ (s1(x
```

:= *v1*), *h1*)) (*fvA* *Q* − {*x*})
  (*s2*(*x* := *v2*), *h2*), (*s2*(*x* := *v2*), *h2*) ⊨ *P* (*s2′*, *h2′*) ∈ *upperize* (Σ (*s2*(*x* := *v2*), *h2*)) (*fvA* *Q* − {*x*})
  using Σ′*-def* **by** *auto*

then obtain *s1″ s2″* **where** *agrees* (*fvA* *Q* − {*x*}) *s1″ s1′* (*s1″*, *h1′*) ∈ Σ (*s1*(*x* := *v1*), *h1*)
  agrees (*fvA* *Q* − {*x*}) *s2″ s2′* (*s2″*, *h2′*) ∈ Σ (*s2*(*x* := *v2*), *h2*)
  using *in-upperize* **by** (*metis* (*no-types*, *lifting*))

moreover have (*s1*(*x* := *v1*), *h1*), (*s2*(*x* := *v2*), *h2*) ⊨ *P*
proof −
  have *v1* = *v1′*
  using ‹(*s1*(*x* := *v1′*), *h1*), (*s2*(*x* := *v2′*), *h2*) ⊨ *P*› *always-sat-refl assms*(*4*) *r*(*1*) *unambiguous-def* **by** *blast*
  moreover have *v2* = *v2′*
  using ‹(*s1*(*x* := *v1′*), *h1*), (*s2*(*x* := *v2′*), *h2*) ⊨ *P*› *always-sat-refl assms*(*4*) *r*(*3*) *sat-comm-aux unambiguous-def* **by** *blast*
  ultimately show *?thesis*
    by (*simp add*: ‹(*s1*(*x* := *v1′*), *h1*), (*s2*(*x* := *v2′*), *h2*) ⊨ *P*›)
qed
then have *pair-sat* (Σ (*s1*(*x* := *v1*), *h1*)) (Σ (*s2*(*x* := *v2*), *h2*)) *Q*
  using *asm0* **by** *simp*
then have (*s1″*, *h1′*), (*s2″*, *h2′*) ⊨ *Q*
  using *calculation*(*2*) *calculation*(*4*) *pair-sat-def* **by** *blast*
moreover have *agrees* (*fvA* *Q*) *s1″* (*s1′*(*x* := *s1″ x*))
proof (*rule agreesI*)
  fix *y* **assume** *y* ∈ *fvA* *Q*
  then show *s1″ y* = (*s1′*(*x* := *s1″ x*)) *y*
    apply (*cases x* = *y*)
     apply *auto*[*1*]
    by (*metis* (*mono-tags*, *lifting*) *DiffI agrees-def calculation*(*1*) *fun-upd-other singleton-iff*)
qed
moreover have *agrees* (*fvA* *Q*) *s2″* (*s2′*(*x* := *s2″ x*))
proof (*rule agreesI*)
  fix *y* **assume** *y* ∈ *fvA* *Q*
  then show *s2″ y* = (*s2′*(*x* := *s2″ x*)) *y*
    apply (*cases x* = *y*)
     apply *auto*[*1*]
    by (*metis* (*mono-tags*, *lifting*) *DiffI agrees-def calculation*(*3*) *fun-upd-other singleton-iff*)
qed
ultimately have (*s1′*(*x* := *s1″ x*), *h1′*), (*s2′*(*x* := *s2″ x*), *h2′*) ⊨ *Q*
  by (*meson agrees-same sat-comm*)
then show (*s1′*, *h1′*), (*s2′*, *h2′*) ⊨ *Exists x Q*
  using *hyper-sat.simps*(*7*) **by** *blast*
qed
qed

qed

4.4.14 While loops

inductive *leads-to-loop* **where**
leads-to-loop $b\ I\ \Sigma\ \sigma\ \sigma$
| $[\![$ *leads-to-loop* $b\ I\ \Sigma\ \sigma\ \sigma'$; *bdenot* b (*fst* σ') ; $\sigma'' \in \Sigma\ \sigma'$ $]\!] \Longrightarrow$ *leads-to-loop* $b\ I\ \Sigma\ \sigma\ \sigma''$

definition *leads-to-loop-set* **where**
leads-to-loop-set $b\ I\ \Sigma\ \sigma$ = { σ' |σ'. *leads-to-loop* $b\ I\ \Sigma\ \sigma\ \sigma'$}

definition *trans-*Σ **where**
trans-Σ $b\ I\ \Sigma\ \sigma$ = *Set.filter* ($\lambda\sigma$. $\neg$ *bdenot* b (*fst* σ)) (*leads-to-loop-set* $b\ I\ \Sigma\ \sigma$)

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

lemma *safe-while-None*:
assumes $\bigwedge\sigma\ m.\ \sigma, \sigma \models$ *And* I (*Bool* b) $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe* n (*None* :: ('*i*, '*a*, *nat*) *cont*) $C\ \sigma$ ($\Sigma\ \sigma$)
and $\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models$ *And* I (*Bool* b) $\Longrightarrow$ *pair-sat* ($\Sigma\ \sigma$) ($\Sigma\ \sigma'$) I
and (s, h), (s, h) $\models I$
and *leads-to-loop* $b\ I\ \Sigma\ \sigma$ (s, h)
and *bounded* h
shows *safe* n (*None* :: ('*i*, '*a*, *nat*) *cont*) (*Cwhile* $b\ C$) (s, h) (*trans-*Σ $b\ I\ \Sigma\ \sigma$)
using *assms*
proof (*induct* n *arbitrary*: $s\ h$)
let *?S* = *trans-*Σ $b\ I\ \Sigma\ \sigma$
case (*Suc* n)
show *?case*
proof (*rule safeNoneI*)
show *no-abort* (*None* :: ('*i*, '*a*, *nat*) *cont*) (*Cwhile* $b\ C$) $s\ h$
using *abort-while-cases no-abortNoneI* **by** *blast*
fix $H\ hf\ C'\ s'\ h'$
assume *asm0*: *Some* H = *Some* $h \oplus$ *Some* hf $\wedge$ *full-ownership* (*get-fh* H) $\wedge$ *no-guard* H $\wedge$ *red* (*Cwhile* $b\ C$) (s, *FractionalHeap.normalize* (*get-fh* H)) C' (s', h')
show $\exists h''\ H'$. *full-ownership* (*get-fh* H') $\wedge$ *no-guard* H' $\wedge$ h' = *FractionalHeap.normalize* (*get-fh* H')
$\wedge$ *Some* H' = *Some* $h'' \oplus$ *Some* hf $\wedge$ *safe* n (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (s', h'') (*trans-*Σ $b\ I\ \Sigma\ \sigma$)
proof (*rule red-while-cases*)
show *red* (*Cwhile* $b\ C$) (s, *FractionalHeap.normalize* (*get-fh* H)) C' (s', h')
using *asm0* **by** *linarith*
assume *asm1*: C' = *Cif* b (*Cseq* C (*Cwhile* $b\ C$)) *Cskip* (s', h') = (s, *FractionalHeap.normalize* (*get-fh* H))
have *safe* n (*None* :: ('*i*, '*a*, *nat*) *cont*) C' (s, h) *?S*

proof (*cases n*)
case (*Suc k*)
have *safe* (*Suc k*) (*None* :: ($'i$, $'a$, *nat*) *cont*) (*Cif b* (*Cseq C* (*Cwhile b C*)) *Cskip*) (*s*, *h*) *?S*
proof (*rule if-safe*)
have ¬ *bdenot b s* $\Longrightarrow$ (*s*, *h*) ∈ *?S*
by (*metis CollectI Suc.prems*(*4*) *asm1*(*2*) *fst-eqD leads-to-loop-set-def member-filter trans-*Σ*-def*)
then show ¬ *bdenot b s* $\Longrightarrow$ *safe k* (*None* :: ($'i$, $'a$, *nat*) *cont*) *Cskip* (*s*, *h*) (*trans-*Σ *b I* Σ σ)
by (*metis Pair-inject asm1*(*2*) *safe-skip*)
assume *asm2*: *bdenot b s*
then have (*s*, *h*), (*s*, *h*) ⊨ *And I* (*Bool b*)
by (*simp add*: *Suc.prems*(*3*))
then have *r*: *safe* (*Suc n*) (*None* :: ($'i$, $'a$, *nat*) *cont*) *C* (*s*, *h*) (Σ (*s*, *h*))
using *Suc.prems*(*1*)
by (*metis Suc.prems*(*5*) *snd-eqD*)

show *safe k* (*None* :: ($'i$, $'a$, *nat*) *cont*) (*Cseq C* (*Cwhile b C*)) (*s*, *h*) (*trans-*Σ *b I* Σ σ)
proof (*rule seq-safe*)
show *safe k* (*None* :: ($'i$, $'a$, *nat*) *cont*) *C* (*s*, *h*) (*Set.filter* (*bounded* ∘ *snd*) (Σ (*s*, *h*)))
apply (*rule restrict-safe-to-bounded*)
using *Suc Suc-n-not-le-n nat-le-linear r safe-smaller* **apply** *metis*
by (*simp add*: *Suc.prems*(*5*))
fix *m* s' h' **assume** *asm3*: $m \le k \wedge (s', h') \in$ *Set.filter* (*bounded* ∘ *snd*) (Σ (*s*, *h*))
have *safe n* (*None* :: ($'i$, $'a$, *nat*) *cont*) (*Cwhile b C*) (s', h') (*trans-*Σ *b I* Σ σ)
proof (*rule Suc.hyps*)
show *leads-to-loop b I* Σ σ (s', h')
by (*metis Suc.prems*(*4*) *asm2 asm3 fst-conv leads-to-loop.simps member-filter*)
show (s', h'), (s', h') ⊨ *I*
using ‹(*s*, *h*), (*s*, *h*) ⊨ *And I* (*Bool b*)› *asm3 assms*(*2*) *pair-satE*
by (*metis member-filter*)
show $\bigwedge\sigma$. σ, σ ⊨ *And I* (*Bool b*) $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe n* (*None* :: ($'i$, $'a$, *nat*) *cont*) *C* σ (Σ σ)
by (*meson Suc.prems*(*1*) *Suc-n-not-le-n nat-le-linear safe-smaller*)
show *bounded* h'
using *asm3* **by** *auto*
qed (*auto simp add*: *assms*)
then show *safe m* (*None* :: ($'i$, $'a$, *nat*) *cont*) (*Cwhile b C*) (s', h') (*trans-*Σ *b I* Σ σ)
using *Suc asm3 le-SucI safe-smaller* **by** *blast*
qed

qed
then show *?thesis*
using *Suc asm1*(*1*) **by** *blast*
qed (*simp*)
then show $\exists h'' H'$. *full-ownership* (*get-fh* H') $\wedge$
no-guard $H' \wedge h' =$ *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* $H' =$ *Some* $h'' \oplus$ *Some hf* $\wedge$ *safe n None* C' (s', h'') (*trans*-Σ *b I* Σ σ)
using *asm0 asm1*(*2*) **by** *blast*
qed
qed (*simp-all*)
qed (*simp*)

lemma *safe-while-Some*:
assumes $\bigwedge \sigma$ *m*. σ, $\sigma \models$ *And I* (*Bool b*) $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe n* (*Some* Γ) *C* σ (Σ σ)
and $\bigwedge \sigma\ \sigma'$. σ, $\sigma' \models$ *And I* (*Bool b*) $\Longrightarrow$ *pair-sat* (Σ σ) (Σ σ') *I*
and (*s*, *h*), (*s*, *h*) $\models$ *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)
using *assms*
proof (*induct n arbitrary*: *s h*)
let *?S* = *trans*-Σ *b I* Σ σ
case (*Suc n*)
show *?case*
proof (*rule safeSomeI*)
show *no-abort* (*Some* Γ) (*Cwhile b C*) *s h*
using *abort-while-cases no-abortSomeI* **by** *blast*
fix *H hf* C' s' h' *hj v0*
assume *asm0*: *Some H* = *Some h* $\oplus$ *Some hj* $\oplus$ *Some hf* $\wedge$
full-ownership (*get-fh H*) $\wedge$ *semi-consistent* Γ *v0 H* $\wedge$ *sat-inv s hj* Γ $\wedge$ *red* (*Cwhile b C*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) C' (s', h')
show $\exists h''\ H'\ hj'$. *full-ownership* (*get-fh* H') $\wedge$ *semi-consistent* Γ *v0* $H' \wedge$ *sat-inv* s' hj' Γ $\wedge$
$h' =$ *FractionalHeap.normalize* (*get-fh* H') $\wedge$ *Some* $H' =$ *Some* $h'' \oplus$ *Some* $hj' \oplus$ *Some hf* $\wedge$ *safe n* (*Some* Γ) C' (s', h'') (*trans*-Σ *b I* Σ σ)
proof (*rule red-while-cases*)
show *red* (*Cwhile b C*) (*s*, *FractionalHeap.normalize* (*get-fh H*)) C' (s', h')
using *asm0* **by** *linarith*
assume *asm1*: C' = *Cif b* (*Cseq C* (*Cwhile b C*)) *Cskip* (s', h') = (*s*, *FractionalHeap.normalize* (*get-fh H*))
have *safe n* (*Some* Γ) C' (*s*, *h*) *?S*
proof (*cases n*)
case (*Suc k*)
have *safe* (*Suc k*) (*Some* Γ) (*Cif b* (*Cseq C* (*Cwhile b C*)) *Cskip*) (*s*, *h*) *?S*
proof (*rule if-safe*)
have $\neg$ *bdenot b s* $\Longrightarrow$ (*s*, *h*) $\in$ *?S*
by (*metis CollectI Suc.prems*(*4*) *asm1*(*2*) *fst-eqD leads-to-loop-set-def member-filter trans*-Σ-*def*)

then show $\neg$ *bdenot b s* $\Longrightarrow$ *safe k* (*Some* Γ) *Cskip* (*s*, *h*) (*trans-*Σ *b I* Σ σ)
by (*metis Pair-inject asm1*(*2*) *safe-skip*)
assume *asm2*: *bdenot b s*
then have (*s*, *h*), (*s*, *h*) $\models$ *And I* (*Bool b*)
by (*simp add*: *Suc.prems*(*3*))
moreover have *bounded h*
apply (*rule bounded-smaller*[*of H*])
using *asm0 full-ownership-then-bounded* **apply** *fastforce*
using *asm0 plus-asso*[*of Some h Some hj Some hf*] **unfolding** *larger-def* **by** *auto*

ultimately have *r*: *safe* (*Suc n*) (*Some* Γ) *C* (*s*, *h*) (Σ (*s*, *h*))
using *Suc.prems*(*1*)[*of* (*s*, *h*)] **by** *fastforce*

show *safe k* (*Some* Γ) (*Cseq C* (*Cwhile b C*)) (*s*, *h*) (*trans-*Σ *b I* Σ σ)
proof (*rule seq-safe*)
show *safe k* (*Some* Γ) *C* (*s*, *h*) (Σ (*s*, *h*))
by (*metis Suc Suc-n-not-le-n nat-le-linear r safe-smaller*)
fix *m s' h'* **assume** *asm3*: $m \leq k \wedge (s', h') \in \Sigma\ (s, h)$
have *safe n* (*Some* Γ) (*Cwhile b C*) (*s'*, *h'*) (*trans-*Σ *b I* Σ σ)
proof (*rule Suc.hyps*)
show *leads-to-loop b I* Σ σ (*s'*, *h'*)
by (*metis Suc.prems*(*4*) *asm2 asm3 fst-conv leads-to-loop.intros*(*2*))
show (*s'*, *h'*), (*s'*, *h'*) $\models$ *I*
using ‹(*s*, *h*), (*s*, *h*) $\models$ *And I* (*Bool b*)› *asm3 assms*(*2*) *pair-satE* **by** *blast*
show $\bigwedge\sigma$. σ, σ $\models$ *And I* (*Bool b*) $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe n* (*Some* Γ) *C* σ (Σ σ)
by (*meson Suc.prems*(*1*) *Suc-n-not-le-n nat-le-linear safe-smaller*)
qed (*auto simp add*: *assms*)
then show *safe m* (*Some* Γ) (*Cwhile b C*) (*s'*, *h'*) (*trans-*Σ *b I* Σ σ)
using *Suc asm3 le-SucI safe-smaller* **by** *blast*
qed
qed
then show *?thesis*
using *Suc asm1*(*1*) **by** *blast*
qed (*simp*)
then show $\exists$ *h'' H' hj'*.
full-ownership (*get-fh H'*) $\wedge$
semi-consistent Γ *v0 H'* $\wedge$
sat-inv s' hj' Γ $\wedge$ *h'* = *FractionalHeap.normalize* (*get-fh H'*) $\wedge$ *Some H'* = *Some h''* $\oplus$ *Some hj'* $\oplus$ *Some hf* $\wedge$ *safe n* (*Some* Γ) *C'* (*s'*, *h''*) (*trans-*Σ *b I* Σ σ)
using *asm0 asm1*(*2*) **by** *blast*
qed
qed (*simp-all*)
qed (*simp*)

lemma *safe-while*:

fixes Δ :: $('i, 'a, nat)$ *cont*
assumes $\bigwedge\sigma\ m.\ \sigma, \sigma \models And\ I\ (Bool\ b) \Longrightarrow bounded\ (snd\ \sigma) \Longrightarrow safe\ n\ \Delta\ C\ \sigma\ (\Sigma\ \sigma)$
and $\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models And\ I\ (Bool\ b) \Longrightarrow pair\text{-}sat\ (\Sigma\ \sigma)\ (\Sigma\ \sigma')\ I$
and $(s, h), (s, h) \models I$
and $leads\text{-}to\text{-}loop\ b\ I\ \Sigma\ \sigma\ (s, h)$
and *bounded h*
shows $safe\ n\ \Delta\ (Cwhile\ b\ C)\ (s, h)\ (trans\text{-}\Sigma\ b\ I\ \Sigma\ \sigma)$
apply $(cases\ \Delta)$
using *assms safe-while-None* **apply** *blast*
using *assms safe-while-Some* **by** *blast*

lemma *leads-to-sat-inv-unary*:
assumes $leads\text{-}to\text{-}loop\ b\ I\ \Sigma\ \sigma\ \sigma'$
and $\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models (And\ I\ (Bool\ b)) \Longrightarrow pair\text{-}sat\ (\Sigma\ \sigma)\ (\Sigma\ \sigma')\ I$
and $\sigma, \sigma \models I$
shows $\sigma', \sigma' \models I$
using *assms*
proof (*induct arbitrary*: *rule*: *leads-to-loop.induct*)
case $(2\ b\ I\ \Sigma\ \sigma 0\ \sigma 1\ \sigma 2)$
then have $pair\text{-}sat\ (\Sigma\ \sigma 1)\ (\Sigma\ \sigma 1)\ I$
by (*metis hyper-sat.simps*(*1*) *hyper-sat.simps*(*3*) *prod.collapse*)
then show *?case*
using *2.hyps*(*4*) *pair-sat-def* **by** *blast*
qed (*simp*)

theorem *while-rule2*:
fixes Δ :: $('i, 'a, nat)$ *cont*
assumes *unary I*
and $hoare\text{-}triple\text{-}valid\ \Delta\ (And\ I\ (Bool\ b))\ C\ I$
shows $hoare\text{-}triple\text{-}valid\ \Delta\ I\ (Cwhile\ b\ C)\ (And\ I\ (Bool\ (Bnot\ b)))$
proof –
obtain Σ **where** *asm0*: $\bigwedge\sigma\ n.\ \sigma, \sigma \models (And\ I\ (Bool\ b)) \Longrightarrow bounded\ (snd\ \sigma) \Longrightarrow safe\ n\ \Delta\ C\ \sigma\ (\Sigma\ \sigma)$
and $\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models (And\ I\ (Bool\ b)) \Longrightarrow pair\text{-}sat\ (\Sigma\ \sigma)\ (\Sigma\ \sigma')\ I$
using *assms*(*2*) *hoare-triple-validE* **by** *blast*
let $?\Sigma = trans\text{-}\Sigma\ b\ I\ \Sigma$
show *?thesis*
proof (*rule hoare-triple-validI-bounded*)

show $\bigwedge s\ h\ s'\ h'.\ (s, h), (s', h') \models I \Longrightarrow pair\text{-}sat\ (?\Sigma\ (s, h))\ (?\Sigma\ (s', h'))\ (And\ I\ (Bool\ (Bnot\ b)))$
proof –
fix *s1 h1 s2 h2* **assume** *asm0*: $(s1, h1), (s2, h2) \models I$
show $pair\text{-}sat\ (trans\text{-}\Sigma\ b\ I\ \Sigma\ (s1, h1))\ (trans\text{-}\Sigma\ b\ I\ \Sigma\ (s2, h2))\ (And\ I\ (Bool\ (Bnot\ b)))$
proof (*rule pair-satI*)
fix *s1' h1' s2' h2'*
assume *asm1*: $(s1', h1') \in trans\text{-}\Sigma\ b\ I\ \Sigma\ (s1, h1) \land (s2', h2') \in trans\text{-}\Sigma$

$b\ I\ \Sigma\ (s2,\ h2)$
then obtain *leads-to-loop* $b\ I\ \Sigma\ (s1,\ h1)\ (s1',\ h1')\ \neg$ *bdenot* $b\ s1'$
leads-to-loop $b\ I\ \Sigma\ (s2,\ h2)\ (s2',\ h2')\ \neg$ *bdenot* $b\ s2'$
using *trans-Σ-def leads-to-loop-set-def*
by (*metis fst-conv mem-Collect-eq member-filter*)
then have $(s1',\ h1'),\ (s1',\ h1') \models I \wedge (s2',\ h2'),\ (s2',\ h2') \models I$
by (*meson* ‹$\bigwedge\sigma'\ \sigma.\ \sigma,\ \sigma' \models$ *And* I (*Bool* b) $\Longrightarrow$ *pair-sat* $(\Sigma\ \sigma)\ (\Sigma\ \sigma')\ I$› *always-sat-refl asm0 leads-to-sat-inv-unary sat-comm-aux*)
then show $(s1',\ h1'),\ (s2',\ h2') \models$ *And* I (*Bool* (*Bnot* b))
by (*metis* ‹¬ *bdenot* $b\ s1'$› ‹¬ *bdenot* $b\ s2'$› *assms*(1) *bdenot.simps*(3) *hyper-sat.simps*(1) *hyper-sat.simps*(3) *unaryE*)
qed
qed
fix $s\ h\ n$
assume *asm1*: $(s,\ h),\ (s,\ h) \models I$ *bounded* h

show *safe* $n\ \Delta$ (*Cwhile* $b\ C$) $(s,\ h)$ (*trans-Σ* $b\ I\ \Sigma\ (s,\ h)$)
proof (*rule safe-while*)
show $\bigwedge\sigma\ \sigma'.\ \sigma,\ \sigma' \models$ *And* I (*Bool* b) $\Longrightarrow$ *pair-sat* $(\Sigma\ \sigma)\ (\Sigma\ \sigma')\ I$
by (*simp add*: ‹$\bigwedge\sigma'\ \sigma.\ \sigma,\ \sigma' \models$ *And* I (*Bool* b) $\Longrightarrow$ *pair-sat* $(\Sigma\ \sigma)\ (\Sigma\ \sigma')\ I$›)
show $(s,\ h),\ (s,\ h) \models I$
using *asm1* **by** *auto*
show *leads-to-loop* $b\ I\ \Sigma\ (s,\ h)\ (s,\ h)$
by (*simp add*: *leads-to-loop.intros*(1))
show $\bigwedge\sigma\ m.\ \sigma,\ \sigma \models$ *And* I (*Bool* b) $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe* $n\ \Delta\ C\ \sigma\ (\Sigma\ \sigma)$
by (*simp add*: *asm0 asm1*(2))
show *bounded* h
using *asm1*(2) **by** *blast*
qed
qed
qed

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 $0\ b\ I\ \Sigma\ \sigma = \{\sigma\}$
| *iterate-sigma* (*Suc* n) $b\ I\ \Sigma\ \sigma = (\bigcup\sigma' \in$ *Set.filter* ($\lambda\sigma.$ *bdenot* b (*fst* σ)) (*iterate-sigma* $n\ b\ I\ \Sigma\ \sigma$). $\Sigma\ \sigma'$)

lemma *union-of-iterate-sigma-is-leads-to-loop-set*:
assumes *leads-to-loop* $b\ I\ \Sigma\ \sigma\ \sigma'$
shows $\sigma' \in (\bigcup n.$ *iterate-sigma* $n\ b\ I\ \Sigma\ \sigma)$
using *assms*
proof (*induct rule*: *leads-to-loop.induct*)
case $(1\ b\ I\ \Sigma\ \sigma)$
have $\sigma \in$ *iterate-sigma* $0\ b\ I\ \Sigma\ \sigma$

by *simp*
then show *?case*
by *blast*
next
case (*2 b I* Σ σ σ' σ'')
then obtain *n* **where** $\sigma' \in$ *iterate-sigma n b I* Σ σ **by** *blast*
then have $\sigma'' \in$ *iterate-sigma* (*Suc n*) *b I* Σ σ **using** *2* **by** *auto*
then show *?case* **by** *blast*
qed

lemma *trans-included*:
trans-Σ *b I* Σ $\sigma \subseteq$ *Set.filter* ($\lambda\sigma$. $\neg$ *bdenot b* (*fst* σ)) ($\bigcup$*n. iterate-sigma n b I* Σ σ)
proof
fix *x* **assume** $x \in$ *trans-*Σ *b I* Σ σ
then have $\neg$ *bdenot b* (*fst x*) $\wedge$ *leads-to-loop b I* Σ σ *x*
by (*simp add*: *leads-to-loop-set-def trans-*Σ*-def*)
then show $x \in$ *Set.filter* ($\lambda\sigma$. $\neg$ *bdenot b* (*fst* σ)) ($\bigcup$*n. iterate-sigma n b I* Σ σ)
by (*metis member-filter union-of-iterate-sigma-is-leads-to-loop-set*)
qed

lemma *iterate-sigma-low-all-sat-I-and-low*:
assumes $\bigwedge\sigma\ \sigma'$. σ, $\sigma' \models$ (*And I* (*Bool b*)) $\Longrightarrow$ *pair-sat* (Σ σ) (Σ σ') (*And I* (*Low b*))
and $\sigma 1$, $\sigma 2 \models I$
and *bdenot b* (*fst* $\sigma 1$) = *bdenot b* (*fst* $\sigma 2$)
shows *pair-sat* (*iterate-sigma n b I* Σ $\sigma 1$) (*iterate-sigma n b I* Σ $\sigma 2$) (*And I* (*Low b*))
using *assms*
proof (*induct n*)
case *0*
then show *?case*
by (*metis* (*mono-tags*, *lifting*) *hyper-sat.simps*(*3*) *hyper-sat.simps*(*5*) *iterate-sigma.simps*(*1*) *pair-satI prod.exhaust-sel singletonD*)
next
case (*Suc n*)
show *?case*
proof (*rule pair-satI*)
fix *s1 h1 s2 h2*
assume *asm0*: (*s1*, *h1*) $\in$ *iterate-sigma* (*Suc n*) *b I* Σ $\sigma 1$ $\wedge$ (*s2*, *h2*) $\in$ *iterate-sigma* (*Suc n*) *b I* Σ $\sigma 2$
then obtain $\sigma 1'$ $\sigma 2'$ **where** *bdenot b* (*fst* $\sigma 1'$) *bdenot b* (*fst* $\sigma 2'$)
$\sigma 1' \in$ *iterate-sigma n b I* Σ $\sigma 1$ $\sigma 2' \in$ *iterate-sigma n b I* Σ $\sigma 2$
(*s1*, *h1*) $\in \Sigma$ $\sigma 1'$ (*s2*, *h2*) $\in \Sigma$ $\sigma 2'$
by *auto*
then have *pair-sat* (*iterate-sigma n b I* Σ $\sigma 1$) (*iterate-sigma n b I* Σ $\sigma 2$) (*And I* (*Low b*))
using *Suc.hyps*

using *Suc.prems*(*3*) *assms*(*1*) *assms*(*2*) **by** *blast*
moreover have *pair-sat* (Σ $\sigma 1'$) (Σ $\sigma 2'$) (*And I* (*Low b*))
proof (*rule Suc.prems*)
show $\sigma 1'$, $\sigma 2'$ $\models$ *And I* (*Bool b*)
by (*metis* ‹$\sigma 1' \in$ *iterate-sigma n b I* Σ $\sigma 1$› ‹$\sigma 2' \in$ *iterate-sigma n b I* Σ $\sigma 2$› ‹*bdenot b* (*fst* $\sigma 1'$)› ‹*bdenot b* (*fst* $\sigma 2'$)› *calculation hyper-sat.simps*(*1*) *hyper-sat.simps*(*3*) *pair-sat-def prod.exhaust-sel*)
qed
ultimately show (*s1*, *h1*), (*s2*, *h2*) $\models$ *And I* (*Low b*)
using ‹(*s1*, *h1*) $\in$ Σ $\sigma 1'$› ‹(*s2*, *h2*) $\in$ Σ $\sigma 2'$› *pair-sat-def* **by** *blast*
qed
qed

lemma *iterate-empty-later-empty*:
assumes *iterate-sigma n b I* Σ σ = {}
and $m \geq n$
shows *iterate-sigma m b I* Σ σ = {}
using *assms*
proof (*induct* $m - n$ *arbitrary*: *n m*)
case (*Suc k*)
then obtain *mm* **where** *m* = *Suc mm*
by (*metis iterate-sigma.elims zero-diff*)
then have *iterate-sigma mm b I* Σ σ = {}
by (*metis Suc.hyps*(*1*) *Suc.hyps*(*2*) *Suc.prems*(*1*) *Suc.prems*(*2*) *Suc-le-mono diff-Suc-Suc diff-diff-cancel diff-le-self*)
then show *?case*
using ‹*m* = *Suc mm*› **by** *force*
qed (*simp*)

lemma *all-same*:
assumes $\bigwedge\sigma\ \sigma'$. σ, σ' $\models$ (*And I* (*Bool b*)) $\Longrightarrow$ *pair-sat* (Σ σ) (Σ σ') (*And I* (*Low b*))
and $\sigma 1$, $\sigma 2$ $\models$ *I*
and *bdenot b* (*fst* $\sigma 1$) = *bdenot b* (*fst* $\sigma 2$)
and *x1* $\in$ *iterate-sigma n b I* Σ $\sigma 1$
and *x2* $\in$ *iterate-sigma n b I* Σ $\sigma 2$
shows *bdenot b* (*fst x1*) = *bdenot b* (*fst x2*)
proof −
have *x1*, *x2* $\models$ (*And I* (*Low b*))
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *iterate-sigma-low-all-sat-I-and-low pair-sat-def* **by** *blast*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *hyper-sat.simps*(*3*) *hyper-sat.simps*(*5*) *surjective-pairing*)
qed

lemma *non-empty-at-most-once*:
assumes $\bigwedge\sigma\ \sigma'$. σ, σ' $\models$ (*And I* (*Bool b*)) $\Longrightarrow$ *pair-sat* (Σ σ) (Σ σ') (*And I* (*Low b*))

and $\sigma, \sigma \models I$
and $\mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\mathit{iterate\text{-}sigma}\ \mathit{n1}\ b\ I\ \Sigma\ \sigma) \neq \{\}$
and $\mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\mathit{iterate\text{-}sigma}\ \mathit{n2}\ b\ I\ \Sigma\ \sigma) \neq \{\}$
shows $\mathit{n1} = \mathit{n2}$
proof $-$
let $?n = \mathit{min}\ \mathit{n1}\ \mathit{n2}$
obtain σ' **where** $\sigma' \in \mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\mathit{iterate\text{-}sigma}\ ?n\ b\ I\ \Sigma\ \sigma)$
by $(\mathit{metis}\ \mathit{assms}(3)\ \mathit{assms}(4)\ \mathit{equals0I}\ \mathit{min.orderE}\ \mathit{min\text{-}def})$
then have $\neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma')$
by *fastforce*
moreover have $\mathit{pair\text{-}sat}\ (\mathit{iterate\text{-}sigma}\ ?n\ b\ I\ \Sigma\ \sigma)\ (\mathit{iterate\text{-}sigma}\ ?n\ b\ I\ \Sigma\ \sigma)\ (\mathit{And}\ I\ (\mathit{Low}\ b))$
using $\mathit{assms}(1)\ \mathit{assms}(2)\ \mathit{assms}(3)\ \mathit{iterate\text{-}sigma\text{-}low\text{-}all\text{-}sat\text{-}I\text{-}and\text{-}low}$ **by** *blast*
then have r: $\bigwedge x.\ x \in \mathit{iterate\text{-}sigma}\ ?n\ b\ I\ \Sigma\ \sigma \Longrightarrow \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ x)$
by $(\mathit{metis}\ \langle\sigma' \in \mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\mathit{iterate\text{-}sigma}\ (\mathit{min}\ \mathit{n1}\ \mathit{n2})\ b\ I\ \Sigma\ \sigma)\rangle\ \mathit{all\text{-}same}\ \mathit{assms}(1)\ \mathit{assms}(2)\ \mathit{member\text{-}filter})$
then have $\mathit{iterate\text{-}sigma}\ (\mathit{Suc}\ ?n)\ b\ I\ \Sigma\ \sigma = \{\}$ **by** *auto*
then have $\neg\ (\mathit{n1} > ?n) \wedge \neg\ (\mathit{n2} > ?n)$ **using** $\mathit{iterate\text{-}empty\text{-}later\text{-}empty}[\mathit{of}\ \mathit{Suc}\ ?n\ b\ I\ \Sigma\ \sigma]$
assms **by** $(\mathit{metis}\ (\mathit{no\text{-}types},\ \mathit{lifting})\ \mathit{Set.filter\text{-}def}\ \mathit{empty\text{-}Collect\text{-}eq}\ \mathit{empty\text{-}def}\ \mathit{le\text{-}simps}(3)\ \mathit{mem\text{-}Collect\text{-}eq})$
then show *?thesis* **by** *linarith*
qed

lemma *one-non-empty-union*:
assumes $\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models (\mathit{And}\ I\ (\mathit{Bool}\ b)) \Longrightarrow \mathit{pair\text{-}sat}\ (\Sigma\ \sigma)\ (\Sigma\ \sigma')\ (\mathit{And}\ I\ (\mathit{Low}\ b))$
and $\sigma, \sigma \models I$
and $\mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\mathit{iterate\text{-}sigma}\ k\ b\ I\ \Sigma\ \sigma) \neq \{\}$
shows $\mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\bigcup n.\ \mathit{iterate\text{-}sigma}\ n\ b\ I\ \Sigma\ \sigma) = \mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\mathit{iterate\text{-}sigma}\ k\ b\ I\ \Sigma\ \sigma)$
proof
show $\mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\mathit{iterate\text{-}sigma}\ k\ b\ I\ \Sigma\ \sigma) \subseteq \mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\bigcup n.\ \mathit{iterate\text{-}sigma}\ n\ b\ I\ \Sigma\ \sigma)$
by *auto*
show $\mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\bigcup n.\ \mathit{iterate\text{-}sigma}\ n\ b\ I\ \Sigma\ \sigma) \subseteq \mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\mathit{iterate\text{-}sigma}\ k\ b\ I\ \Sigma\ \sigma)$
proof
fix x **assume** $x \in \mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\bigcup n.\ \mathit{iterate\text{-}sigma}\ n\ b\ I\ \Sigma\ \sigma)$
then obtain k' **where** $x \in \mathit{iterate\text{-}sigma}\ k'\ b\ I\ \Sigma\ \sigma\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ x)$
by *auto*
then have $x \in \mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\mathit{iterate\text{-}sigma}\ k'\ b\ I\ \Sigma\ \sigma)$
by *fastforce*
then have $k = k'$
using $\mathit{non\text{-}empty\text{-}at\text{-}most\text{-}once}\ \mathit{assms}(1)\ \mathit{assms}(2)\ \mathit{assms}(3)$ **by** *blast*
then show $x \in \mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\mathit{iterate\text{-}sigma}\ k\ b\ I\ \Sigma\ \sigma)$
using $\langle x \in \mathit{Set.filter}\ (\lambda\sigma.\ \neg\ \mathit{bdenot}\ b\ (\mathit{fst}\ \sigma))\ (\mathit{iterate\text{-}sigma}\ k'\ b\ I\ \Sigma\ \sigma)\rangle$ **by**

blast
  qed
qed

definition *not-set* **where**
  not-set b S = *Set.filter* ($\lambda\sigma$. ¬ *bdenot b* (*fst* σ)) *S*

lemma *union-exists-at-some-point-exactly*:
  assumes $\bigwedge\sigma\ \sigma'$. σ, σ' ⊨ (*And I* (*Bool b*)) ⟹ *pair-sat* (Σ σ) (Σ σ') (*And I* (*Low b*))
      and $\sigma 1$, $\sigma 2$ ⊨ *I*
      and *bdenot b* (*fst* $\sigma 1$) = *bdenot b* (*fst* $\sigma 2$)
      and *Set.filter* ($\lambda\sigma$. ¬ *bdenot b* (*fst* σ)) ($\bigcup$*n*. *iterate-sigma n b I* Σ $\sigma 1$) ≠ {}
      and *Set.filter* ($\lambda\sigma$. ¬ *bdenot b* (*fst* σ)) ($\bigcup$*n*. *iterate-sigma n b I* Σ $\sigma 2$) ≠ {}
    shows ∃ *k*. *not-set b* ($\bigcup$*n*. *iterate-sigma n b I* Σ $\sigma 1$) = *not-set b* (*iterate-sigma k b I* Σ $\sigma 1$) ∧ *not-set b* ($\bigcup$*n*. *iterate-sigma n b I* Σ $\sigma 2$) = *not-set b* (*iterate-sigma k b I* Σ $\sigma 2$)
proof −
  obtain *k1* **where** *Set.filter* ($\lambda\sigma$. ¬ *bdenot b* (*fst* σ)) (*iterate-sigma k1 b I* Σ $\sigma 1$) ≠ {}
    using *assms*(*4*) **by** *fastforce*
  moreover obtain *k2* **where** *Set.filter* ($\lambda\sigma$. ¬ *bdenot b* (*fst* σ)) (*iterate-sigma k2 b I* Σ $\sigma 2$) ≠ {}
    using *assms*(*5*) **by** *fastforce*

  show *?thesis*
  proof (*cases k1* ≤ *k2*)
    case *True*
    then have *iterate-sigma k1 b I* Σ $\sigma 2$ ≠ {}
     by (*metis* (*no-types*, *lifting*) *Collect-cong Set.filter-def* ‹*Set.filter* ($\lambda\sigma$. ¬ *bdenot b* (*fst* σ)) (*iterate-sigma k2 b I* Σ $\sigma 2$) ≠ {}› *empty-def iterate-empty-later-empty mem-Collect-eq*)
   then obtain $\sigma 1'$ $\sigma 2'$ **where** $\sigma 1'$ ∈ *Set.filter* ($\lambda\sigma$. ¬ *bdenot b* (*fst* σ)) (*iterate-sigma k1 b I* Σ $\sigma 1$) ∧ $\sigma 2'$ ∈ *iterate-sigma k1 b I* Σ $\sigma 2$
      using *calculation* **by** *blast*
    then have ¬ *bdenot b* (*fst* $\sigma 1'$)
      by *fastforce*
    moreover have *pair-sat* (*iterate-sigma k1 b I* Σ $\sigma 1$) (*iterate-sigma k1 b I* Σ $\sigma 2$) (*And I* (*Low b*))
       using *assms*(*1*) *assms*(*2*) *assms*(*3*) *iterate-sigma-low-all-sat-I-and-low* **by** *blast*
    then have *r*: $\bigwedge$*x1 x2*. *x1* ∈ *iterate-sigma k1 b I* Σ $\sigma 1$ ∧ *x2* ∈ *iterate-sigma k1 b I* Σ $\sigma 2$ ⟹ *bdenot b* (*fst x1*) ⟷ *bdenot b* (*fst x2*)
     by (*metis* (*no-types*, *opaque-lifting*) *eq-fst-iff hyper-sat.simps*(*3*) *hyper-sat.simps*(*5*) *pair-sat-def*)
    then have ¬ *bdenot b* (*fst* $\sigma 2'$)
      by (*metis* ‹$\sigma 1'$ ∈ *Set.filter* ($\lambda\sigma$. ¬ *bdenot b* (*fst* σ)) (*iterate-sigma k1 b I* Σ $\sigma 1$) ∧ $\sigma 2'$ ∈ *iterate-sigma k1 b I* Σ $\sigma 2$› *member-filter*)
    then have $\bigwedge$*x1*. *x1* ∈ *iterate-sigma k1 b I* Σ $\sigma 1$ ⟹ ¬ *bdenot b* (*fst x1*)

using ‹$\sigma 1' \in$ *Set.filter* $(\lambda\sigma.\ \neg$ *bdenot b* (*fst* σ)) (*iterate-sigma k1 b I* $\Sigma\ \sigma 1$) $\wedge\ \sigma 2' \in$ *iterate-sigma k1 b I* $\Sigma\ \sigma 2$› *r* **by** *blast*
then have *iterate-sigma* (*Suc k1*) *b I* $\Sigma\ \sigma 1 = \{\}$ **by** *auto*
moreover have $\bigwedge x2.\ x2 \in$ *iterate-sigma k1 b I* $\Sigma\ \sigma 2 \Longrightarrow \neg$ *bdenot b* (*fst x2*)
by (*metis* ‹$\sigma 1' \in$ *Set.filter* $(\lambda\sigma.\ \neg$ *bdenot b* (*fst* σ)) (*iterate-sigma k1 b I* Σ $\sigma 1$) $\wedge\ \sigma 2' \in$ *iterate-sigma k1 b I* $\Sigma\ \sigma 2$› *member-filter r*)
then have *iterate-sigma* (*Suc k1*) *b I* $\Sigma\ \sigma 2 = \{\}$ **by** *auto*
then have $k1 = k2$
using *True* ‹*Set.filter* $(\lambda\sigma.\ \neg$ *bdenot b* (*fst* σ)) (*iterate-sigma k2 b I* $\Sigma\ \sigma 2$) $\neq$ $\{\}$› *dual-order.antisym*[*of k1 k2*]
ex-in-conv iterate-empty-later-empty[*of - b I* $\Sigma\ \sigma 2$] *member-filter not-less-eq-eq*
by *metis*
moreover have *Set.filter* $(\lambda\sigma.\ \neg$ *bdenot b* (*fst* σ)) ($\bigcup n.$ *iterate-sigma n b I* Σ $\sigma 1$) = *Set.filter* $(\lambda\sigma.\ \neg$ *bdenot b* (*fst* σ)) (*iterate-sigma k1 b I* $\Sigma\ \sigma 1$)
using *one-non-empty-union*[*of I b* $\Sigma\ \sigma 1$]
using ‹*Set.filter* $(\lambda\sigma.\ \neg$ *bdenot b* (*fst* σ)) (*iterate-sigma k1 b I* $\Sigma\ \sigma 1$) $\neq \{\}$› *always-sat-refl assms*(*1*) *assms*(*2*) **by** *blast*
moreover have *Set.filter* $(\lambda\sigma.\ \neg$ *bdenot b* (*fst* σ)) ($\bigcup n.$ *iterate-sigma n b I* Σ $\sigma 2$) = *Set.filter* $(\lambda\sigma.\ \neg$ *bdenot b* (*fst* σ)) (*iterate-sigma k1 b I* $\Sigma\ \sigma 2$)
using *one-non-empty-union*[*of I b* $\Sigma\ \sigma 2$]
using ‹*Set.filter* $(\lambda\sigma.\ \neg$ *bdenot b* (*fst* σ)) (*iterate-sigma k2 b I* $\Sigma\ \sigma 2$) $\neq \{\}$› *always-sat-refl assms*(*1*) *assms*(*2*) *calculation*(*3*) *sat-comm* **by** *blast*
ultimately show *?thesis*
by (*metis not-set-def*)
next
case *False*
then have *iterate-sigma k2 b I* $\Sigma\ \sigma 1 \neq \{\}$
by (*metis* (*no-types*, *lifting*) *Collect-cong Set.filter-def calculation empty-def iterate-empty-later-empty linorder-le-cases mem-Collect-eq*)
then obtain $\sigma 1'\ \sigma 2'$ **where** $\sigma 1' \in$ *iterate-sigma k2 b I* $\Sigma\ \sigma 1 \wedge \sigma 2' \in$ *not-set b* (*iterate-sigma k2 b I* $\Sigma\ \sigma 2$)
by (*metis* ‹*Set.filter* $(\lambda\sigma.\ \neg$ *bdenot b* (*fst* σ)) (*iterate-sigma k2 b I* $\Sigma\ \sigma 2$) $\neq$ $\{\}$› *ex-in-conv not-set-def*)
then have $\neg$ *bdenot b* (*fst* $\sigma 2'$)
using *not-set-def* **by** *fastforce*
then have $\neg$ *bdenot b* (*fst* $\sigma 1'$)
by (*metis* ‹$\sigma 1' \in$ *iterate-sigma k2 b I* $\Sigma\ \sigma 1 \wedge \sigma 2' \in$ *not-set b* (*iterate-sigma k2 b I* $\Sigma\ \sigma 2$)› *all-same assms*(*1*) *assms*(*2*) *assms*(*3*) *member-filter not-set-def*)
then have $\bigwedge x1.\ x1 \in$ *iterate-sigma k2 b I* $\Sigma\ \sigma 1 \Longrightarrow \neg$ *bdenot b* (*fst x1*)
using ‹$\sigma 1' \in$ *iterate-sigma k2 b I* $\Sigma\ \sigma 1 \wedge \sigma 2' \in$ *not-set b* (*iterate-sigma k2 b I* $\Sigma\ \sigma 2$)› *all-same always-sat-refl assms*(*1*) *assms*(*2*) **by** *blast*
then have *iterate-sigma* (*Suc k2*) *b I* $\Sigma\ \sigma 1 = \{\}$ **by** *auto*
moreover have $\bigwedge x2.\ x2 \in$ *iterate-sigma k2 b I* $\Sigma\ \sigma 2 \Longrightarrow \neg$ *bdenot b* (*fst x2*)
using ‹$\neg$ *bdenot b* (*fst* $\sigma 1'$)› ‹$\sigma 1' \in$ *iterate-sigma k2 b I* $\Sigma\ \sigma 1 \wedge \sigma 2' \in$ *not-set b* (*iterate-sigma k2 b I* $\Sigma\ \sigma 2$)› *all-same assms*(*1*) *assms*(*2*) *assms*(*3*) **by** *blast*
then have *iterate-sigma* (*Suc k2*) *b I* $\Sigma\ \sigma 2 = \{\}$ **by** *auto*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *Collect-empty-eq False Set.filter-def* ‹*Set.filter* $(\lambda\sigma.\ \neg$ *bdenot b* (*fst* σ)) (*iterate-sigma k1 b I* $\Sigma\ \sigma 1$) $\neq \{\}$› *calculation empty-iff*

iterate-empty-later-empty not-less-eq-eq)
qed
qed

theorem *while-rule1*:
fixes Δ :: ($'i$, $'a$, *nat*) *cont*
assumes *hoare-triple-valid* Δ (*And I* (*Bool b*)) *C* (*And I* (*Low b*))
shows *hoare-triple-valid* Δ (*And I* (*Low b*)) (*Cwhile b C*) (*And I* (*Bool* (*Bnot b*)))
proof −
obtain Σ **where** *asm0*: $\bigwedge\sigma$ *n*. σ, σ $\models$ (*And I* (*Bool b*)) $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe n* Δ *C* σ (Σ σ)
and $\bigwedge\sigma\ \sigma'$. σ, σ' $\models$ (*And I* (*Bool b*)) $\Longrightarrow$ *pair-sat* (Σ σ) (Σ σ') (*And I* (*Low b*))
using *assms*(*1*) *hoare-triple-validE* **by** *blast*
let *?*Σ = $\lambda\sigma$. *not-set b* ($\bigcup$*n*. *iterate-sigma n b I* Σ σ)
show *?thesis*
proof (*rule hoare-triple-validI-bounded*)

show $\bigwedge$*s h s' h'*. (*s*, *h*), (*s'*, *h'*) $\models$ *And I* (*Low b*) $\Longrightarrow$ *pair-sat* (*?*Σ (*s*, *h*)) (*?*Σ (*s'*, *h'*)) (*And I* (*Bool* (*Bnot b*)))
proof −
fix *s1 h1 s2 h2* **assume** *asm0*: (*s1*, *h1*), (*s2*, *h2*) $\models$ *And I* (*Low b*)
then have *asm0-bis*: (*s1*, *h1*), (*s2*, *h2*) $\models$ *I* $\wedge$ *bdenot b* (*fst* (*s1*, *h1*)) = *bdenot b* (*fst* (*s2*, *h2*)) **by** *auto*

show *pair-sat* (*not-set b* ($\bigcup$*n*. *iterate-sigma n b I* Σ (*s1*, *h1*))) (*not-set b* ($\bigcup$*n*. *iterate-sigma n b I* Σ (*s2*, *h2*))) (*And I* (*Bool* (*Bnot b*)))
proof (*rule pair-satI*)
fix *s1' h1' s2' h2'*
assume *asm1*: (*s1'*, *h1'*) $\in$ *not-set b* ($\bigcup$*n*. *iterate-sigma n b I* Σ (*s1*, *h1*)) $\wedge$ (*s2'*, *h2'*) $\in$ *not-set b* ($\bigcup$*n*. *iterate-sigma n b I* Σ (*s2*, *h2*))
then obtain *k* **where** *not-set b* ($\bigcup$*n*. *iterate-sigma n b I* Σ (*s1*, *h1*)) = *not-set b* (*iterate-sigma k b I* Σ (*s1*, *h1*))
not-set b ($\bigcup$*n*. *iterate-sigma n b I* Σ (*s2*, *h2*)) = *not-set b* (*iterate-sigma k b I* Σ (*s2*, *h2*))
using *union-exists-at-some-point-exactly*[*of I b* Σ (*s1*, *h1*) (*s2*, *h2*)] *asm0-bis not-set-def*
using ‹$\bigwedge\sigma'\ \sigma$. σ, σ' $\models$ *And I* (*Bool b*) $\Longrightarrow$ *pair-sat* (Σ σ) (Σ σ') (*And I* (*Low b*))› **by** *blast*
moreover have *pair-sat* (*iterate-sigma k b I* Σ (*s1*, *h1*)) (*iterate-sigma k b I* Σ (*s2*, *h2*)) (*And I* (*Low b*))
using ‹$\bigwedge\sigma'\ \sigma$. σ, σ' $\models$ *And I* (*Bool b*) $\Longrightarrow$ *pair-sat* (Σ σ) (Σ σ') (*And I* (*Low b*))› *asm0-bis iterate-sigma-low-all-sat-I-and-low* **by** *blast*
ultimately show (*s1'*, *h1'*), (*s2'*, *h2'*) $\models$ *And I* (*Bool* (*Bnot b*))
by (*metis* (*no-types*, *lifting*) *asm1 bdenot.simps*(*3*) *fst-conv hyper-sat.simps*(*1*) *hyper-sat.simps*(*3*) *member-filter not-set-def pair-satE*)
qed

qed

fix *s h n*
assume *asm1*: $(s, h), (s, h) \models$ *And I* (*Low b*) *bounded h*

have *safe n* Δ (*Cwhile b C*) (s, h) (*trans*-Σ *b I* Σ (s, h))
proof (*rule safe-while*)
show $\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models$ *And I* (*Bool b*) $\Longrightarrow$ *pair-sat* $(\Sigma\ \sigma)$ $(\Sigma\ \sigma')$ *I*
by (*meson* ‹$\bigwedge\sigma'\ \sigma.\ \sigma, \sigma' \models$ *And I* (*Bool b*) $\Longrightarrow$ *pair-sat* $(\Sigma\ \sigma)$ $(\Sigma\ \sigma')$ (*And I* (*Low b*))› *hyper-sat.simps*(*3*) *pair-sat-def*)
show $(s, h), (s, h) \models I$
using *asm1* **by** *auto*
show *leads-to-loop b I* Σ (s, h) (s, h)
by (*simp add*: *leads-to-loop.intros*(*1*))
show $\bigwedge\sigma\ m.\ \sigma, \sigma \models$ *And I* (*Bool b*) $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe n* Δ *C* σ $(\Sigma\ \sigma)$
by (*simp add*: *asm0 asm1*)
show *bounded h*
by (*simp add*: *asm1*(*2*))
qed
then show *safe n* Δ (*Cwhile b C*) (s, h) (*not-set b* ($\bigcup n.$ *iterate-sigma n b I* Σ (s, h)))
by (*simp add*: *not-set-def safe-larger-set trans-included*)
qed
qed

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*
by (*simp add*: *assms entails-def*)

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

show *hoare-triple-valid* Δ (*And P* (*Low* (*Band e e'*))) (*Cwhile* (*Band e e'*) *C*) (*And P* (*Bool* (*Bnot* (*Band e e'*))))
proof (*rule while-rule1*)

show *hoare-triple-valid* Δ (*And P* (*Bool* (*Band e e'*))) *C* (*And P* (*Low* (*Band e e'*)))
by (*simp add*: *assms*(*5*))
qed

show *hoare-triple-valid* Δ (*And P* (*Bool* (*Bnot* (*Band e e'*)))) (*Cwhile e C*) (*And* (*Star P' R*) (*Bool* (*Bnot e*)))
proof (*rule consequence-rule*)
show *hoare-triple-valid* Δ (*Star P' R*) (*Cwhile e C*) (*Star* (*And P'* (*Bool* (*Bnot e*))) *R*)
proof (*rule frame-rule*)
show *precise P'* $\vee$ *precise R*
by (*simp add*: *assms*(*6*))
show *disjoint* (*fvA R*) (*wrC* (*Cwhile e C*))
by (*simp add*: *assms*(*3*) *disjoint-def*)
show *hoare-triple-valid* Δ *P'* (*Cwhile e C*) (*And P'* (*Bool* (*Bnot e*)))
proof (*rule while-rule2*)
show *hoare-triple-valid* Δ (*And P'* (*Bool e*)) *C P'*
by (*simp add*: *assms*(*4*))
show *unary P'* **using** *assms*(*2*) **by** *auto*
qed
qed
show *entails* (*And P* (*Bool* (*Bnot* (*Band e e'*)))) (*Star P' R*)
using *assms*(*1*) *entails-def hyper-sat.simps*(*3*) **by** *blast*
show *entails* (*Star* (*And P'* (*Bool* (*Bnot e*))) *R*) (*And* (*Star P' R*) (*Bool* (*Bnot e*)))
proof (*rule entails-smallerI*)
fix *s1 h1 s2 h2*
assume *asm0*: (*s1*, *h1*), (*s2*, *h2*) $\models$ *Star* (*And P'* (*Bool* (*Bnot e*))) *R*
then obtain *hp1 hr1 hp2 hr2* **where** *Some h1* = *Some hp1* $\oplus$ *Some hr1* *Some h2* = *Some hp2* $\oplus$ *Some hr2*
(*s1*, *hp1*), (*s2*, *hp2*) $\models$ *And P'* (*Bool* (*Bnot e*)) (*s1*, *hr1*), (*s2*, *hr2*) $\models$ *R*
using *hyper-sat.simps*(*4*) **by** *blast*
then show (*s1*, *h1*), (*s2*, *h2*) $\models$ *And* (*Star P' R*) (*Bool* (*Bnot e*))
by *fastforce*
qed
qed
qed

4.4.15 CommCSL is sound

theorem *soundness*:
assumes $\Delta \vdash \{P\}\ C\ \{Q\}$
shows $\Delta \models \{P\}\ C\ \{Q\}$
using *assms*
proof (*induct rule*: *CommCSL.induct*)
case (*RuleAtomicShared* Γ *f* α *sact uact J P Q x C map-to-arg sarg ms map-to-multiset* π)
then show *?case* **using** *atomic-rule-shared* **by** *blast*

qed (*simp-all add*: *rule-skip assign-rule new-rule write-rule read-rule share-rule atomic-rule-unique*
rule-par if1-rule if2-rule seq-rule frame-rule consequence-rule existential-rule while-rule1 while-rule2)

4.5 Corollaries

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

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

theorem *safety*:
assumes $\mathit{hoare\text{-}triple\text{-}valid}\ (\mathit{None} :: ('i, 'a, \mathit{nat})\ \mathit{cont})\ P\ C\ Q$
and $(s1, h1), (s2, h2) \models P$

and $\mathit{Some}\ H1 = \mathit{Some}\ h1 \oplus \mathit{Some}\ hf1 \wedge \mathit{full\text{-}ownership}\ (\mathit{get\text{-}fh}\ H1) \wedge \mathit{no\text{-}guard}\ H1$
— extend h1 to a normal state H1 without guards

and $\mathit{Some}\ H2 = \mathit{Some}\ h2 \oplus \mathit{Some}\ hf2 \wedge \mathit{full\text{-}ownership}\ (\mathit{get\text{-}fh}\ H2) \wedge \mathit{no\text{-}guard}\ H2$
— extend h2 to a normal state H2 without guards

shows $\bigwedge\sigma'\ C'.\ \mathit{red\text{-}rtrans}\ C\ (s1, \mathit{normalize}\ (\mathit{get\text{-}fh}\ H1))\ C'\ \sigma' \Longrightarrow \neg\ \mathit{aborts}\ C'\ \sigma'$
and $\bigwedge\sigma'\ C'.\ \mathit{red\text{-}rtrans}\ C\ (s2, \mathit{normalize}\ (\mathit{get\text{-}fh}\ H2))\ C'\ \sigma' \Longrightarrow \neg\ \mathit{aborts}\ C'\ \sigma'$
and $\bigwedge\sigma 1'\ \sigma 2'.\ \mathit{red\text{-}rtrans}\ C\ (s1, \mathit{normalize}\ (\mathit{get\text{-}fh}\ H1))\ \mathit{Cskip}\ \sigma 1'$
$\Longrightarrow \mathit{red\text{-}rtrans}\ C\ (s2, \mathit{normalize}\ (\mathit{get\text{-}fh}\ H2))\ \mathit{Cskip}\ \sigma 2'$
$\Longrightarrow (\exists h1'\ h2'\ H1'\ H2'.\ \mathit{no\text{-}guard}\ H1' \wedge \mathit{full\text{-}ownership}\ (\mathit{get\text{-}fh}\ H1') \wedge \mathit{snd}\ \sigma 1' = \mathit{normalize}\ (\mathit{get\text{-}fh}\ H1') \wedge \mathit{Some}\ H1' = \mathit{Some}\ h1' \oplus \mathit{Some}\ hf1$
$\wedge\ \mathit{no\text{-}guard}\ H2' \wedge \mathit{full\text{-}ownership}\ (\mathit{get\text{-}fh}\ H2') \wedge \mathit{snd}\ \sigma 2' = \mathit{normalize}\ (\mathit{get\text{-}fh}\ H2') \wedge \mathit{Some}\ H2' = \mathit{Some}\ h2' \oplus \mathit{Some}\ hf2$
$\wedge\ (\mathit{fst}\ \sigma 1', h1'), (\mathit{fst}\ \sigma 2', h2') \models Q)$
proof –
obtain Σ **where** *asm0*: $\bigwedge\sigma\ n.\ \sigma, \sigma \models P \Longrightarrow \mathit{bounded}\ (\mathit{snd}\ \sigma) \Longrightarrow \mathit{safe}\ n\ (\mathit{None} :: ('i, 'a, \mathit{nat})\ \mathit{cont})\ C\ \sigma\ (\Sigma\ \sigma)$
$\bigwedge\sigma\ \sigma'.\ \sigma, \sigma' \models P \Longrightarrow \mathit{pair\text{-}sat}\ (\Sigma\ \sigma)\ (\Sigma\ \sigma')\ Q$
using *assms*(*1*) *hoare-triple-validE* **by** *blast*
then have $\mathit{pair\text{-}sat}\ (\Sigma\ (s1, h1))\ (\Sigma\ (s2, h2))\ Q$
using *assms*(*2*) **by** *blast*
moreover have *bounded h1*
by (*metis assms*(*3*) *bounded-smaller-sum full-ownership-then-bounded get-fh.simps*)
then have $\bigwedge n.\ \mathit{safe}\ n\ (\mathit{None} :: ('i, 'a, \mathit{nat})\ \mathit{cont})\ C\ (s1, h1)\ (\Sigma\ (s1, h1))$
using *always-sat-refl asm0*(*1*) *assms*(*2*)
by (*metis snd-conv*)

then show $\bigwedge\sigma'\ C'$. *red-rtrans C* (*s1*, *FractionalHeap.normalize* (*get-fh H1*)) C' $\sigma' \Longrightarrow \neg$ *aborts* $C'\ \sigma'$
proof –
fix $\sigma'\ C'$
assume *red-rtrans C* (*s1*, *FractionalHeap.normalize* (*get-fh H1*)) $C'\ \sigma'$
then show $\neg$ *aborts* $C'\ \sigma'$
using *safe-atomic*[*of C* (*s1*, *FractionalHeap.normalize* (*get-fh H1*)) $C'\ \sigma'$ *s1 FractionalHeap.normalize* (*get-fh H1*) *fst* σ' *snd* σ']
by (*metis* ‹$\bigwedge n$. *safe n None C* (*s1*, *h1*) (Σ (*s1*, *h1*))› *assms*(*3*) *denormalize-properties*(*4*) *prod.exhaust-sel*)
qed
moreover have $\bigwedge n$. *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C* (*s2*, *h2*) (Σ (*s2*, *h2*))
using *always-sat-refl asm0*(*1*) *assms*(*2*) *sat-comm-aux*
by (*metis assms*(*4*) *bounded-smaller-sum full-ownership-then-bounded get-fh.elims snd-conv*)
then show $\bigwedge\sigma'\ C'$. *red-rtrans C* (*s2*, *FractionalHeap.normalize* (*get-fh H2*)) C' $\sigma' \Longrightarrow \neg$ *aborts* $C'\ \sigma'$
proof –
fix $\sigma'\ C'$
assume *red-rtrans C* (*s2*, *FractionalHeap.normalize* (*get-fh H2*)) $C'\ \sigma'$
then show $\neg$ *aborts* $C'\ \sigma'$
using *safe-atomic*[*of C* (*s2*, *FractionalHeap.normalize* (*get-fh H2*)) $C'\ \sigma'$ *s2 FractionalHeap.normalize* (*get-fh H2*) *fst* σ' *snd* σ']
by (*metis* ‹$\bigwedge n$. *safe n None C* (*s2*, *h2*) (Σ (*s2*, *h2*))› *assms*(*4*) *denormalize-properties*(*4*) *prod.exhaust-sel*)
qed
fix $\sigma 1'$
assume *red-rtrans C* (*s1*, *FractionalHeap.normalize* (*get-fh H1*)) *Cskip* $\sigma 1'$
then obtain *h1' H1'* **where** *r1*: *Some H1'* = *Some h1'* $\oplus$ *Some hf1 snd* $\sigma 1'$ = *FractionalHeap.normalize* (*get-fh H1'*)
no-guard H1' $\wedge$ *full-ownership* (*get-fh H1'*) (*fst* $\sigma 1'$, *h1'*) $\in \Sigma$ (*s1*, *h1*)
using *safe-atomic*[*of C* (*s1*, *FractionalHeap.normalize* (*get-fh H1*)) *Cskip* $\sigma 1'$ *s1 - fst* $\sigma 1'$ *snd* $\sigma 1'$ *h1* Σ (*s1*, *h1*) *H1 hf1*]
by (*metis* ‹$\bigwedge n$. *safe n None C* (*s1*, *h1*) (Σ (*s1*, *h1*))› *assms*(*3*) *denormalize-properties*(*4*) *surjective-pairing*)
fix $\sigma 2'$
assume *red-rtrans C* (*s2*, *FractionalHeap.normalize* (*get-fh H2*)) *Cskip* $\sigma 2'$
then obtain *h2' H2'* **where** *r2*: *Some H2'* = *Some h2'* $\oplus$ *Some hf2 snd* $\sigma 2'$ = *FractionalHeap.normalize* (*get-fh H2'*)
no-guard H2' $\wedge$ *full-ownership* (*get-fh H2'*) (*fst* $\sigma 2'$, *h2'*) $\in \Sigma$ (*s2*, *h2*)
using *safe-atomic*[*of C* (*s2*, *FractionalHeap.normalize* (*get-fh H2*)) *Cskip* $\sigma 2'$ *s2 - fst* $\sigma 2'$ *snd* $\sigma 2'$ *h2* Σ (*s2*, *h2*) *H2 hf2*]
by (*metis* ‹$\bigwedge n$. *safe n None C* (*s2*, *h2*) (Σ (*s2*, *h2*))› *assms*(*4*) *denormalize-properties*(*4*) *surjective-pairing*)
then have (*fst* $\sigma 1'$, *h1'*), (*fst* $\sigma 2'$, *h2'*) $\models Q$
using *calculation*(*1*) *pair-satE r1*(*4*) **by** *blast*
then show $\exists$ *h1' h2' H1' H2'*.
no-guard H1' $\wedge$
full-ownership (*get-fh H1'*) $\wedge$

snd σ1′ = FractionalHeap.normalize (get-fh H1′) ∧
Some H1′ = Some h1′ ⊕ Some hf1 ∧
no-guard H2′ ∧
full-ownership (get-fh H2′) ∧ snd σ2′ = FractionalHeap.normalize (get-fh H2′) ∧ Some H2′ = Some h2′ ⊕ Some hf2 ∧ (fst σ1′, h1′), (fst σ2′, h2′) ⊨ Q
using *r1 r2* **by** *blast*
qed

lemma *neutral-add*:
Some h = Some h ⊕ Some (Map.empty, None, (λ-. None))
proof −
have *h ## (Map.empty, None, (λ-. None))*
by (*metis compatibleI compatible-fract-heapsI empty-heap-def fst-conv get-fh.elims get-gs.simps get-gu.simps option.distinct(1) snd-conv*)
then obtain *x* **where** *Some x = Some h ⊕ Some (Map.empty, None, (λ-. None))*
by *simp*
moreover have *x = h*
by (*metis (no-types, lifting) addition-cancellative calculation decompose-guard-remove-easy fst-eqD get-gs.simps get-gu.simps no-guard-def no-guards-remove prod.sel(2) simpler-asso*)
ultimately show *?thesis* **by** *blast*
qed

This second corollary considers that the heap h1 is the only execution that matters, and thus it ignores any frame. It corresponds to Corollary 4.5 in the paper.

corollary *safety-no-frame*:
assumes *hoare-triple-valid (None :: ('i, 'a, nat) cont) P C Q*
and *(s1, H1), (s2, H2) ⊨ 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 −
have *Some H1 = Some H1 ⊕ Some (Map.empty, None, (λ-. None))*
using *neutral-add* **by** *blast*
moreover have *Some H2 = Some H2 ⊕ Some (Map.empty, None, (λ-. None))*
using *neutral-add* **by** *blast*

show $\bigwedge\sigma'$ C'. *red-rtrans C* (*s1*, *FractionalHeap.normalize* (*get-fh H1*)) C' σ' $\Longrightarrow$ $\neg$ *aborts* C' σ'
using *always-sat-refl-aux assms*(*1*) *assms*(*2*) *assms*(*3*) *calculation safety*(*2*) **by** *blast*
show $\bigwedge\sigma'$ C'. *red-rtrans C* (*s2*, *FractionalHeap.normalize* (*get-fh H2*)) C' σ' $\Longrightarrow$ $\neg$ *aborts* C' σ'
using ‹*Some H2* = *Some H2* ⊕ *Some* (*Map.empty*, *None*, (λ-. *None*))› *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *calculation safety*(*2*) **by** *blast*
fix $\sigma 1'$ $\sigma 2'$
assume *red-rtrans C* (*s1*, *FractionalHeap.normalize* (*get-fh H1*)) *Cskip* $\sigma 1'$
red-rtrans C (*s2*, *FractionalHeap.normalize* (*get-fh H2*)) *Cskip* $\sigma 2'$

then obtain *h1′ h2′ H1′ H2′* **where** *asm0*: *no-guard H1′* ∧ *full-ownership* (*get-fh H1′*) ∧ *snd* $\sigma 1'$ = *normalize* (*get-fh H1′*) ∧ *Some H1′* = *Some h1′* ⊕ *Some* (*Map.empty*, *None*, (λ-. *None*))
∧ *no-guard H2′* ∧ *full-ownership* (*get-fh H2′*) ∧ *snd* $\sigma 2'$ = *normalize* (*get-fh H2′*) ∧ *Some H2′* = *Some h2′* ⊕ *Some* (*Map.empty*, *None*, (λ-. *None*))
∧ (*fst* $\sigma 1'$, *h1′*), (*fst* $\sigma 2'$, *h2′*) ⊨ *Q*
using *safety*[*of P C Q s1 H1 s2 H2 H1* (*Map.empty*, *None*, (λ-. *None*)) *H2* (*Map.empty*, *None*, (λ-. *None*))] *assms*
by (*metis* (*no-types*, *lifting*) ‹*Some H2* = *Some H2* ⊕ *Some* (*Map.empty*, *None*, (λ-. *None*))› *calculation*)
then have *H1′* = *h1′*
using *addition-cancellative decompose-guard-remove-easy denormalize-properties*(*4*) *denormalize-properties*(*5*)
by (*metis denormalize-def get-gs.simps get-gu.simps prod.exhaust-sel snd-conv*)
moreover have *H2′* = *h2′*
by (*metis asm0 denormalize-properties*(*4*) *denormalize-properties*(*5*) *fst-eqD get-fh.elims no-guard-and-no-heap no-guard-then-smaller-same*)

ultimately show ∃ *H1′ H2′*.
no-guard H1′ ∧
full-ownership (*get-fh H1′*) ∧
snd $\sigma 1'$ = *FractionalHeap.normalize* (*get-fh H1′*) ∧
no-guard H2′ ∧ *full-ownership* (*get-fh H2′*) ∧ *snd* $\sigma 2'$ = *FractionalHeap.normalize* (*get-fh H2′*) ∧ (*fst* $\sigma 1'$, *H1′*), (*fst* $\sigma 2'$, *H2′*) ⊨ *Q*
using *asm0* **by** *blast*
qed

end
theory *NonInterference*
imports *Soundness*
begin

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

fun *low-list* **where**
low-list [] = *Bool Btrue*
| *low-list* (*v* # *q*) = *And* (*LowExp* (*Evar v*)) (*low-list q*)

lemma *low-listE*:
  assumes $(s1, h1), (s2, h2) \models$ *low-list l*
    and $x \in$ *set l*
  shows *s1 x = s2 x*
  using *assms*
proof (*induct l*)
  case (*Cons a l*)
  then show *?case*
  proof (*cases x = a*)
    case *True*
    then have $(s1, h1), (s2, h2) \models$ *LowExp* (*Evar a*)
      using *Cons.prems*(*1*) **by** *auto*
    then show *?thesis*
      by (*simp add*: *True*)
  next
    case *False*
    then show *?thesis*
      using *Cons.hyps Cons.prems*(*1*) *Cons.prems*(*2*) **by** *auto*
  qed
qed (*simp*)

lemma *low-listI*:
  assumes $\bigwedge x.\ x \in$ *set l* $\Longrightarrow$ *s1 x = s2 x*
  shows $(s1, h1), (s2, h2) \models$ *low-list l*
  using *assms*
by (*induct l*) *simp-all*

corollary *non-interference*:
  assumes (*None* :: ('*i*, '*a*, *nat*) *cont*) $\vdash$ {*And P* (*low-list In*)} *C* {*low-list Out*}
    and *red-rtrans C* (*s1*, *normalize* (*get-fh H1*)) *Cskip* (*s1'*, *h1'*)
    and *red-rtrans C* (*s2*, *normalize* (*get-fh H2*)) *Cskip* (*s2'*, *h2'*)
    and $\bigwedge x.\ x \in$ *set In* $\Longrightarrow$ *s1 x = s2 x*
    and $x \in$ *set Out*
    and $(s1, H1), (s2, H2) \models P$
    and *full-ownership* (*get-fh H1*) $\wedge$ *no-guard H1*
    and *full-ownership* (*get-fh H2*) $\wedge$ *no-guard H2*
  shows *s1' x = s2' x*
proof –
  have $\exists$ *H1' H2'*. *no-guard H1'* $\wedge$ *full-ownership* (*get-fh H1'*) $\wedge$ *snd* (*s1'*, *h1'*) = *FractionalHeap.normalize* (*get-fh H1'*) $\wedge$
    no-guard H2' $\wedge$ *full-ownership* (*get-fh H2'*) $\wedge$ *snd* (*s2'*, *h2'*) = *FractionalHeap.normalize* (*get-fh H2'*)
  $\wedge$ (*fst* (*s1'*, *h1'*), *H1'*), (*fst* (*s2'*, *h2'*), *H2'*) $\models$ (*low-list Out* :: ('*i*, '*a*, *nat*) *assertion*)
  proof (*rule safety-no-frame*(*3*))
    show (*None* :: ('*i*, '*a*, *nat*) *cont*) $\models$ {*And P* (*low-list In*)} *C* {*low-list Out*}
      using *assms*(*1*) *soundness* **by** *blast*
    have $(s1, H1), (s2, H2) \models$ *low-list In*

by (*simp add*: *assms*(*4*) *low-listI*)
then show (*s1*, *H1*), (*s2*, *H2*) $\models$ *And P* (*low-list In*)
by (*simp add*: *assms*(*6*))
qed ((*insert assms*; *blast*)+)
then show *?thesis*
by (*metis assms*(*5*) *fst-conv low-listE*)
qed

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 (*rule full-ownershipI*)
fix *l p* **assume** *get-fh* (*heapify h*) *l* = *Some p*
then show *fst p* = *pwrite*
by (*metis apply-opt.elims fst-conv get-fh.elims heapify-def option.sel option.simps*(*3*))
next
show *no-guard* (*heapify h*)
by (*metis addition-cancellative decompose-guard-remove-easy decompose-heap-triple heapify-def neutral-add no-guards-remove snd-conv*)
show *normalize* (*get-fh* (*heapify h*)) = *h*
proof (*rule ext*)
fix *l* **show** *FractionalHeap.normalize* (*get-fh* (*heapify h*)) *l* = *h l*
proof (*cases h l*)
case *None*
then show *?thesis*
by (*metis apply-opt.simps*(*1*) *domIff dom-normalize fst-conv get-fh.simps heapify-def*)
next
case (*Some a*)
then show *?thesis*
by (*simp add*: *FractionalHeap.normalize-eq*(*2*) *heapify-def*)
qed
qed
qed

corollary *non-interference-no-precondition*:
assumes (*None* :: (*'i*, *'a*, *nat*) *cont*) $\vdash$ {*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 $\bigwedge$*x*. *x* $\in$ *set In* $\Longrightarrow$ *s1 x* = *s2 x*
and *x* $\in$ *set Out*
shows *s1' x* = *s2' x*
proof (*rule non-interference*)
show (*None* :: (*'i*, *'a*, *nat*) *cont*) $\vdash$ {*And* (*Bool Btrue*) (*low-list In*)} *C* {*low-list Out*}

using *RuleCons assms*(*1*) *entails-def hyper-sat.simps*(*3*) **by** *blast*
show *red-rtrans C* (*s1*, *FractionalHeap.normalize* (*get-fh* (*heapify h1*))) *Cskip* (*s1′*, *h1′*)
by (*metis assms*(*2*) *heapify-properties*(*3*))
show *red-rtrans C* (*s2*, *FractionalHeap.normalize* (*get-fh* (*heapify h2*))) *Cskip* (*s2′*, *h2′*)
by (*metis assms*(*3*) *heapify-properties*(*3*))
qed (*insert assms heapify-properties*; *auto*)+

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.