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* $f\ h1\ h2$
  ⟨*proof*⟩

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

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

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

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

lemma *add-heaps-asso*:
(*h1* ++ *h2*) ++ *h3* = *h1* ++ (*h2* ++ *h3*)
⟨*proof*⟩

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

lemma *compatible-maps-refl*:
compatible-maps (=) *h h*
⟨*proof*⟩

lemma *map-invo*:
h ++ *h* = *h*
⟨*proof*⟩

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

lemma *commut-charact*:
assumes *compatible-maps* (=) *h1 h2*
shows *h1* ++ *h2* = *h2* ++ *h1*
⟨*proof*⟩

end

1.2 Fractional Permissions

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*} ⟨*proof*⟩

setup-lifting *type-definition-prat*

lift-definition *pwrite* :: *prat* **is** *1* ⟨*proof*⟩

lift-definition *half* :: *prat* **is** *1* / *2* ⟨*proof*⟩

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

lift-definition *pmult* :: *prat* ⇒ *prat* ⇒ *prat* **is** (∗) ⟨*proof*⟩
lift-definition *padd* :: *prat* ⇒ *prat* ⇒ *prat* **is** (+) ⟨*proof*⟩

lift-definition *pdiv* :: *prat* ⇒ *prat* ⇒ *prat* **is** (/) ⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *half-between-0-1*:

pgt pwrite half
⟨*proof*⟩

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

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

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

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

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

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

lemma *greater-sum-both*:
assumes *pgte a* (*padd b c*)
shows ∃ *a1 a2*. *a* = *padd a1 a2* ∧ *pgte a1 b* ∧ *pgte a2 c*
⟨*proof*⟩

lemma *padd-cancellative*:
assumes *a* = *padd x b*
and *a* = *padd y b*
shows *x* = *y*
⟨*proof*⟩

lemma *not-pgte-charact*:
¬ *pgte a b* ⟷ *pgt b a*
⟨*proof*⟩

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

and *pgte c d*
shows *pgt* (*padd a c*) (*padd b d*)
⟨*proof*⟩

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

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

lemma *pmult-special*:
pmult pwrite x = *x*
⟨*proof*⟩

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

lemma *pinv-double-half*:
pmult half (*pinv p*) = *pinv* (*padd p p*)
⟨*proof*⟩

lemma *pinv-inverts*:
assumes *pgte a b*
shows *pgte* (*pinv b*) (*pinv a*)
⟨*proof*⟩

lemma *pinv-pmult-ok*:
pmult p (*pinv p*) = *pwrite*
⟨*proof*⟩

lemma *pinv-pwrite*:
pinv pwrite = *pwrite*
⟨*proof*⟩

lemma *pmin-pmax*:
assumes *pgte x* (*pmin a b*)
shows *x* = *pmin* (*pmax x a*) (*pmax x b*)
⟨*proof*⟩

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

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

end

1.3 Permission Heaps

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* $= 'l \rightharpoonup prat \times 'v$

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

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

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

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

lemma *fadd-options-cancellative*:
assumes *fadd-options a x* = *fadd-options b x*
shows *a* = *b*
⟨*proof*⟩

definition *compatible-fract-heaps* :: $('l, 'v)$ *fract-heap* $\Rightarrow$ $('l, 'v)$ *fract-heap* $\Rightarrow$ *bool*
where
compatible-fract-heaps h h' $\longleftrightarrow$ *compatible-fractions h h'* $\wedge$ *same-values h h'*

lemma *compatible-fract-heapsI*:
assumes $\bigwedge l\ p\ p'.\ h\ l = Some\ p \wedge h'\ l = Some\ p' \Longrightarrow pgte\ pwrite\ (padd\ (fst\ p)\ (fst\ p'))$

and $\bigwedge l\ p\ p'.\ h\ l = \mathit{Some}\ p \wedge h'\ l = \mathit{Some}\ p' \Longrightarrow \mathit{snd}\ p = \mathit{snd}\ p'$
shows *compatible-fract-heaps* $h\ h'$
⟨*proof*⟩

lemma *compatible-fract-heapsE*:
assumes *compatible-fract-heaps* $h\ h'$
and $h\ l = \mathit{Some}\ p \wedge h'\ l = \mathit{Some}\ p'$
shows *pgte pwrite* (*padd* (*fst* p) (*fst* p'))
and $\mathit{snd}\ p = \mathit{snd}\ p'$
⟨*proof*⟩

lemma *compatible-fract-heaps-comm*:
assumes *compatible-fract-heaps* $h\ h'$
shows *compatible-fract-heaps* $h'\ h$
⟨*proof*⟩

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

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

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

lemma *full-ownershipI*:
assumes $\bigwedge l\ p.\ h\ l = \mathit{Some}\ p \Longrightarrow \mathit{fst}\ p = \mathit{pwrite}$
shows *full-ownership* h
⟨*proof*⟩

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* $\Rightarrow$ $('l \rightharpoonup 'v)$ **where**
normalize $h\ l$ = *apply-opt snd* $(h\ l)$

lemma *normalize-eq*:
normalize $h\ l = \mathit{None} \longleftrightarrow h\ l = \mathit{None}$
normalize $h\ l = \mathit{Some}\ v \longleftrightarrow (\exists\, p.\ h\ l = \mathit{Some}\ (p,\ v))$ (**is** $?A \longleftrightarrow ?B$)
⟨*proof*⟩

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

lemma *compatible-then-dom-disjoint*:

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

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

Addition of permission heaps is associative.

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

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

end

1.4 Extended Heaps

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

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

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

We store the initial value with the unique guard

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

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

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

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

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

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

definition *compatible* :: $('i,\ 'a)$ *heap* $\Rightarrow$ $('i,\ 'a)$ *heap* $\Rightarrow$ *bool* (**infixl** ‹##› *60*)
where
h ## $h' \longleftrightarrow$ *compatible-fract-heaps* (*get-fh h*) (*get-fh h'*) $\wedge$ ($\forall k$. *get-gu h k* = *None* $\vee$ *get-gu h' k* = *None*)
$\wedge$ ($\forall p\ p'$. *get-gs h* = *Some p* $\wedge$ *get-gs h'* = *Some p'* $\longrightarrow$ *pgte pwrite* (*padd* (*fst p*) (*fst p'*)))

lemma *compatibleI*:
assumes *compatible-fract-heaps* (*get-fh h*) (*get-fh h'*)
and $\bigwedge k$. *get-gu h k* = *None* $\vee$ *get-gu h' k* = *None*
and $\bigwedge p\ p'$. *get-gs h* = *Some p* $\wedge$ *get-gs h'* = *Some p'* $\Longrightarrow$ *pgte pwrite* (*padd* (*fst p*) (*fst p'*))
shows h ## h'
⟨*proof*⟩

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

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

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

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

Addition of shared guard states is cancellative.

lemma *add-gs-cancellative*:
assumes *add-gs a x* = *add-gs b x*
shows *a* = *b*
⟨*proof*⟩

Addition of shared guard states is commutative.

lemma *add-gs-comm*:

add-gs a b = add-gs b a
⟨*proof*⟩

lemma *compatible-fheaps-comm*:
assumes *compatible-fract-heaps a b*
shows *add-fh a b = add-fh b a*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *compatible-eq*:
Some a ⊕ *Some b = None* ⟷ ¬ *a ## b*
⟨*proof*⟩

lemma *compatible-comm*:
a ## b ⟷ *b ## a*
⟨*proof*⟩

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

Addition of two extended heaps is commutative.

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

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

lemma *plus-extract-point-fh*:
assumes $Some\ x = Some\ a \oplus Some\ b$
and $get\text{-}fh\ a\ l = Some\ pa$
and $get\text{-}fh\ b\ l = Some\ pb$
shows $snd\ pa = snd\ pb \wedge pgte\ pwrite\ (padd\ (fst\ pa)\ (fst\ pb)) \wedge get\text{-}fh\ x\ l = Some\ (padd\ (fst\ pa)\ (fst\ pb),\ snd\ pa)$
⟨*proof*⟩

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

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

Addition of two extended heaps is associative.

lemma *plus-asso*:
$(a \oplus b) \oplus c = a \oplus (b \oplus c)$
⟨*proof*⟩

We define the extension order between extended heaps.

definition *larger* :: $('i, 'a)\ heap \Rightarrow ('i, 'a)\ heap \Rightarrow bool$ (**infixl** ‹$\succeq$› *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*⟩

lemma *comp-smaller*:
assumes $a\ \#\#\ b$
and $Some\ b = Some\ c \oplus Some\ d$
shows $a\ \#\#\ c$
⟨*proof*⟩

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

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

shows *get-gu h k = Some uargs*
⟨*proof*⟩

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

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

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

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

lemma *larger3*:
assumes *Some* x = *Some* $a \oplus$ *Some* $b \oplus$ *Some* c
shows $x \succeq b$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

⟨*proof*⟩

lemma *compatible-last-two*:
assumes $Some\ x = Some\ a \oplus Some\ b \oplus Some\ c$
shows $b\ \#\#\ c$
⟨*proof*⟩

lemma *add-fh-map-empty*:
$add\text{-}fh\ h\ Map.empty = h$
⟨*proof*⟩

definition *bounded* **where**
$bounded\ h \longleftrightarrow (\forall\, l\ p.\ fst\ h\ l = Some\ p \longrightarrow pgte\ pwrite\ (fst\ p))$

lemma *boundedI*:
assumes $\bigwedge l\ p.\ fst\ h\ l = Some\ p \Longrightarrow pgte\ pwrite\ (fst\ p)$
shows $bounded\ h$
⟨*proof*⟩

lemma *boundedE*:
assumes $bounded\ h$
and $fst\ h\ l = Some\ p$
shows $pgte\ pwrite\ (fst\ p)$
⟨*proof*⟩

lemma *bounded-smaller-sum*:
assumes $bounded\ x$
and $Some\ x = Some\ a \oplus Some\ b$
shows $bounded\ a$
⟨*proof*⟩

lemma *bounded-smaller*:
assumes $bounded\ x$
and $x \succeq a$
shows $bounded\ a$
⟨*proof*⟩

lemma *sum-perm-smaller*:
assumes $Some\ x = Some\ a \oplus Some\ b$
and $fst\ a\ l = Some\ (p,\ v)$
shows $\exists\, p'.\ pgte\ p'\ p \wedge fst\ x\ l = Some\ (p',\ v)$
⟨*proof*⟩

lemma *modus-ponens*:
assumes A
and $A \Longrightarrow B$
shows B

$\langle proof \rangle$

lemma *fpdom-inclusion*:
assumes *Some* h' = *Some* $h \oplus$ *Some* r
and *bounded* h'
shows *fpdom* (*fst* h) $\subseteq$ *fpdom* (*fst* h')
$\langle proof \rangle$

lemma *fpdom-dom-disjoint*:
assumes *Some* h = *Some* *h1* $\oplus$ *Some* *h2*
shows *dom* (*fst h1*) $\cap$ *fpdom* (*fst h2*) = $\{\}$
$\langle proof \rangle$

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)
$\langle proof \rangle$

lemma *full-ownership-then-bounded*:
assumes *full-ownership* (*fst* h)
shows *bounded* h
$\langle proof \rangle$

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* $\times$ *normal-heap*

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

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

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

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

2.1.1 Semantics of expressions

Denotational semantics for arithmetic and boolean expressions.

primrec
  $edenot :: exp \Rightarrow store \Rightarrow nat$
where
  $edenot\ (Evar\ v)\ s = s\ v$
| $edenot\ (Enum\ n)\ s = n$
| $edenot\ (Eplus\ e1\ e2)\ s = edenot\ e1\ s + edenot\ e2\ s$

primrec
  $bdenot :: bexp \Rightarrow store \Rightarrow bool$
where
  $bdenot\ (Beq\ e1\ e2)\ s = (edenot\ e1\ s = edenot\ e2\ s)$
| $bdenot\ (Band\ b1\ b2)\ s = (bdenot\ b1\ s \wedge bdenot\ b2\ s)$
| $bdenot\ (Bnot\ b)\ s = (\neg\ bdenot\ b\ s)$
| $bdenot\ Btrue\ \text{-} = True$

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' $\Longrightarrow$ *red* (*Cseq C1 C2*) σ (*Cseq C1' C2*) σ'
| *red-If1* [*intro*]: *bdenot B* (*fst* σ) $\Longrightarrow$ *red* (*Cif B C1 C2*) σ *C1* σ
| *red-If2* [*intro*]: $\neg$ *bdenot B* (*fst* σ) $\Longrightarrow$ *red* (*Cif B C1 C2*) σ *C2* σ
| *red-Atomic*[*intro*]: *red-rtrans C* σ *Cskip* σ' $\Longrightarrow$ *red* (*Catomic C*) σ *Cskip* σ'
| *red-Par1* [*elim*]: *red C1* σ *C1'* σ' $\Longrightarrow$ *red* (*Cpar C1 C2*) σ (*Cpar C1' C2*) σ'
| *red-Par2* [*elim*]: *red C2* σ *C2'* σ' $\Longrightarrow$ *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*) ⟧ $\Longrightarrow$ *red* (*Cassign x E*) σ *Cskip* σ'
| *red-Read*[*intro*]: ⟦ σ = (*s*,*h*); *h*(*edenot E s*) = *Some v*; σ' = (*s*(*x* := *v*), *h*) ⟧ $\Longrightarrow$ *red* (*Cread x E*) σ *Cskip* σ'
| *red-Write*[*intro*]: ⟦ σ = (*s*,*h*); σ' = (*s*, *h*(*edenot E s* ↦ *edenot E' s*)) ⟧ $\Longrightarrow$ *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*)) ⟧ $\Longrightarrow$ *red* (*Calloc x E*) σ *Cskip* σ'
| *red-Free*[*intro*]: ⟦ σ = (*s*,*h*); σ' = (*s*, *h*(*edenot E s* := *None*)) ⟧ $\Longrightarrow$ *red* (*Cdispose E*) σ *Cskip* σ'

| *NoStep*: *red-rtrans C* σ *C* σ
| *OneMoreStep*: ⟦ *red C* σ *C'* σ' ; *red-rtrans C'* σ' *C''* σ'' ⟧ $\Longrightarrow$ *red-rtrans C* σ *C''* σ''

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

2.1.3 Abort semantics

primrec
  accesses :: *cmd* ⇒ *store* ⇒ *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* ∪ *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
$writes\ Cskip\quad s = \{\}$
$|\ writes\ (Cassign\ x\ E)\quad s = \{\}$
$|\ writes\ (Cread\ x\ E)\quad s = \{\}$
$|\ writes\ (Cwrite\ E\ E')\quad s = \{edenot\ E\ s\}$
$|\ writes\ (Calloc\ x\ E)\quad s = \{\}$
$|\ writes\ (Cdispose\ E)\quad s = \{edenot\ E\ s\}$
$|\ writes\ (Cseq\ C1\ C2)\quad s = writes\ C1\ s$
$|\ writes\ (Cpar\ C1\ C2)\quad s = writes\ C1\ s \cup writes\ C2\ s$
$|\ writes\ (Cif\ B\ C1\ C2)\quad s = \{\}$
$|\ writes\ (Cwhile\ B\ C)\quad s = \{\}$
$|\ writes\ (Catomic\ C)\quad s = \{\}$

inductive
$aborts :: cmd \Rightarrow state \Rightarrow bool$
where
$aborts\text{-}Seq[intro]\text{:}\quad aborts\ C1\ \sigma \Longrightarrow aborts\ (Cseq\ C1\ C2)\ \sigma$
$|\ aborts\text{-}Atomic[intro]\text{:}\ [\![\ red\text{-}rtrans\ C\ \sigma\ C'\ \sigma'\ ;\ aborts\ C'\ \sigma'\ ]\!] \Longrightarrow aborts\ (Catomic\ C)\ \sigma$
$|\ aborts\text{-}Par1[intro]\text{:}\quad aborts\ C1\ \sigma \Longrightarrow aborts\ (Cpar\ C1\ C2)\ \sigma$
$|\ aborts\text{-}Par2[intro]\text{:}\quad aborts\ C2\ \sigma \Longrightarrow aborts\ (Cpar\ C1\ C2)\ \sigma$
$|\ aborts\text{-}Read[intro]\text{:}\quad edenot\ E\ (fst\ \sigma) \notin dom\ (snd\ \sigma) \Longrightarrow aborts\ (Cread\ x\ E)\ \sigma$
$|\ aborts\text{-}Write[intro]\text{:}\ edenot\ E\ (fst\ \sigma) \notin dom\ (snd\ \sigma) \Longrightarrow aborts\ (Cwrite\ E\ E')\ \sigma$
$|\ aborts\text{-}Free[intro]\text{:}\quad edenot\ E\ (fst\ \sigma) \notin dom\ (snd\ \sigma) \Longrightarrow aborts\ (Cdispose\ E)\ \sigma$
$|\ aborts\text{-}Race1[intro]\text{:}\quad accesses\ C1\ (fst\ \sigma) \cap writes\ C2\ (fst\ \sigma) \neq \{\} \Longrightarrow aborts\ (Cpar\ C1\ C2)\ \sigma$
$|\ aborts\text{-}Race2[intro]\text{:}\quad writes\ C1\ (fst\ \sigma) \cap accesses\ C2\ (fst\ \sigma) \neq \{\} \Longrightarrow aborts\ (Cpar\ C1\ C2)\ \sigma$

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

2.2 Useful Definitions and Results

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

primrec
$fvE :: exp \Rightarrow var\ set$
where
$fvE\ (Evar\ v) = \{v\}$
$|\ fvE\ (Enum\ n) = \{\}$
$|\ fvE\ (Eplus\ e1\ e2) = (fvE\ e1 \cup fvE\ e2)$

primrec
$fvB :: bexp \Rightarrow var\ set$

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

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

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

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

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

| *subB x E (Bnot b)* = *Bnot (subB x E b)*
| *subB x E Btrue* = *Btrue*

Basic properties of substitutions:

lemma *subE-assign*:
edenot (subE x E e) s = edenot e (s(x := edenot E s))
⟨*proof*⟩

lemma *subB-assign*:
bdenot (subB x E b) s = bdenot b (s(x := edenot E s))
⟨*proof*⟩

inductive-cases *red-skip-cases*: *red Cskip* σ *C'* σ'
inductive-cases *aborts-skip-cases*: *aborts Cskip* σ

lemma *skip-simps*[*simp*]:
$\neg$ *red Cskip* σ *C'* σ'
$\neg$ *aborts Cskip* σ
⟨*proof*⟩

definition
agrees :: *'a set* $\Rightarrow$ (*'a* $\Rightarrow$ *'b*) $\Rightarrow$ (*'a* $\Rightarrow$ *'b*) $\Rightarrow$ *bool*
where
agrees X s s' $\equiv \forall x \in X.\ s\ x = s'\ x$

lemma *agrees-union*:
agrees $(A \cup B)$ *s s'* $\longleftrightarrow$ *agrees A s s'* $\wedge$ *agrees B s s'*
⟨*proof*⟩

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

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

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

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

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

lemma *bexp-agrees*:

$agrees\ (fvB\ B)\ s\ s' \Longrightarrow bdenot\ B\ s = bdenot\ B\ s'$
$\langle proof \rangle$

lemma *red-not-in-fv-not-touched*:
$red\ C\ \sigma\ C'\ \sigma' \Longrightarrow x \notin fvC\ C \Longrightarrow fst\ \sigma\ x = fst\ \sigma'\ x$
$red\text{-}rtrans\ C\ \sigma\ C'\ \sigma' \Longrightarrow x \notin fvC\ C \Longrightarrow fst\ \sigma\ x = fst\ \sigma'\ x$
$\langle proof \rangle$

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

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

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

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

lemma *writes-accesses*: $writes\ C\ s \subseteq accesses\ C\ s$
$\langle proof \rangle$

lemma *accesses-agrees*: $agrees\ (fvC\ C)\ s\ s' \Longrightarrow accesses\ C\ s = accesses\ C\ s'$
$\langle proof \rangle$

lemma *writes-agrees*: $agrees\ (fvC\ C)\ s\ s' \Longrightarrow writes\ C\ s = writes\ C\ s'$
$\langle proof \rangle$

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

corollary *exp-agrees2*[*simp*]:
$x \notin fvE\ E \Longrightarrow edenot\ E\ (s(x := v)) = edenot\ E\ s$
⟨*proof*⟩

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

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

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

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

end

3 CommCSL

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

theory *CommCSL*
imports *Lang StateModel*
begin

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

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

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

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

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

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

3.1 Assertion Language

datatype (*'i*, *'a*, *'v*) *assertion* =
  Bool bexp
  | *Emp*
  | *And* (*'i*, *'a*, *'v*) *assertion* (*'i*, *'a*, *'v*) *assertion*
  | *Star* (*'i*, *'a*, *'v*) *assertion* (*'i*, *'a*, *'v*) *assertion*   (‹- ∗ -› *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* × (*'i*, *'a*) *heap*) $\Rightarrow$ (*store* × (*'i*, *'a*) *heap*) $\Rightarrow$ (*'i*, *'a*, *nat*) *assertion* $\Rightarrow$ *bool* (‹-, - ⊨ -› [*51*, *65*, *66*] *50*) **where**

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

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

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

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

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

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

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

$\mid (s, h), (s', h') \models \mathit{LowExp}\ e \longleftrightarrow \mathit{edenot}\ e\ s = \mathit{edenot}\ e\ s'$

$\mid (s, h), (s', h') \models \mathit{Imp}\ b\ A \longleftrightarrow \mathit{bdenot}\ b\ s = \mathit{bdenot}\ b\ s' \wedge (\mathit{bdenot}\ b\ s \longrightarrow (s, h), (s', h') \models A)$

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

lemma *sat-PreUniqueE*:
assumes $(s, h), (s', h') \models \mathit{PreUniqueGuards}\ \mathit{upre}$
shows $\exists \mathit{uargs}\ \mathit{uargs}'.\ (\forall k.\ \mathit{get\text{-}gu}\ h\ k = \mathit{Some}\ (\mathit{uargs}\ k)) \wedge (\forall k.\ \mathit{get\text{-}gu}\ h'\ k =$

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

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

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*
⟨*proof*⟩

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

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

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

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

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

⟨*proof*⟩

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

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

lemma *subA-assign*:
assumes *collect-existentials A* ∩ *fvE E* = {}
shows (*s*, *h*), (*s'*, *h'*) ⊨ *subA x E A* ⟷ (*s*(*x* := *edenot E s*), *h*), (*s'*(*x* := *edenot E s'*), *h'*) ⊨ *A*
⟨*proof*⟩

lemma *PRE-uniqueI*:
assumes *length uargs* = *length uargs'*
and ⋀*i*. *i* ≥ *0* ∧ *i* < *length uargs'* ⟹ *upre* (*uargs* ! *i*) (*uargs'* ! *i*)
shows *PRE-unique upre uargs uargs'*
⟨*proof*⟩

lemma *PRE-unique-implies-tl*:
assumes *PRE-unique upre* (*ta* # *qa*) (*tb* # *qb*)
shows *PRE-unique upre qa qb*
⟨*proof*⟩

lemma *charact-PRE-unique*:
assumes *PRE-unique* (*Rep-indexed-precondition pre k*) *a b*
shows *PRE-unique* (*Rep-indexed-precondition pre k*) *b a* ∧ *PRE-unique* (*Rep-indexed-precondition pre k*) *a a*
⟨*proof*⟩

lemma *charact-PRE-shared-simpler*:

assumes *PRE-shared-simpler rpre a b*
and *Rep-precondition pre* = *rpre*
shows *PRE-shared-simpler* (*Rep-precondition pre*) *b a* ∧ *PRE-shared-simpler* (*Rep-precondition pre*) *a a*
⟨*proof*⟩

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

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

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

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

lemma *sat-comm-aux*:
$(s, h), (s', h') \models A \Longrightarrow (s', h'), (s, h) \models A$
⟨*proof*⟩

lemma *sat-comm*:
$\sigma, \sigma' \models A \longleftrightarrow \sigma', \sigma \models A$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *preciseE*:
assumes *precise J*
and $H1 \succeq h1 \land H1 \succeq h1' \land H2 \succeq h2 \land H2 \succeq h2'$
and $(s1, h1), (s2, h2) \models J \land (s1, h1'), (s2, h2') \models J$
shows $h1' = h1 \land h2' = h2$
⟨*proof*⟩

definition *unary* **where**
$unary\ J \longleftrightarrow (\forall s\ h\ s'\ h'.\ (s, h), (s, h) \models J \land (s', h'), (s', h') \models J \longrightarrow (s, h), (s', h') \models J)$

lemma *unaryI*:
assumes $\bigwedge s1\ h1\ s2\ h2.\ (s1, h1), (s1, h1) \models J \land (s2, h2), (s2, h2) \models J \Longrightarrow (s1, h1), (s2, h2) \models J$
shows *unary J*
⟨*proof*⟩

lemma *unary-smallerI*:
assumes $\bigwedge \sigma 1\ \sigma 2.\ \sigma 1, \sigma 1 \models J \land \sigma 2, \sigma 2 \models J \Longrightarrow \sigma 1, \sigma 2 \models J$
shows *unary J*
⟨*proof*⟩

lemma *unaryE*:
assumes *unary J*
and $(s, h), (s, h) \models J \land (s', h'), (s', h') \models J$
shows $(s, h), (s', h') \models J$
⟨*proof*⟩

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

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

lemma *sat-points-to*:
assumes $(s, h :: ('i, 'a)\ heap), (s, h) \models PointsTo\ a\ \pi\ e$
shows $get\text{-}fh\ h = [edenot\ a\ s \mapsto (\pi, edenot\ e\ s)]$
⟨*proof*⟩

lemma *unary-inv-then-view*:
assumes *unary J*
shows *unary (View f J e)*

$\langle proof \rangle$

lemma *precise-inv-then-view*:
 assumes *precise J*
 shows *precise* (*View f J e*)
$\langle proof \rangle$

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*
 $\langle proof \rangle$

The following record defines resource contexts (Section 3.5).

record ($'i$, $'a$, $'v$) *single-context* =
 view :: (*loc* $\rightharpoonup$ *val*) $\Rightarrow$ $'v$
 abstract-view :: $'v \Rightarrow 'v$
 saction :: $'v \Rightarrow 'a \Rightarrow 'v$
 uaction :: $'i \Rightarrow 'v \Rightarrow 'a \Rightarrow 'v$
 invariant :: ($'i$, $'a$, $'v$) *assertion*

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

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

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

definition *view-function-of-inv* :: ($'i$, $'a$, *nat*) *single-context* $\Rightarrow$ *bool* **where**
 view-function-of-inv Γ $\longleftrightarrow$ ($\forall$ (*h* :: ($'i$, $'a$) *heap*) (h' :: ($'i$, $'a$) *heap*) *s*. (*s*, *h*), (*s*, *h*) $\models$ *invariant* Γ $\wedge$ ($h' \succeq h$)
 $\longrightarrow$ *view* Γ (*normalize* (*get-fh h*)) = *view* Γ (*normalize* (*get-fh* h')))

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

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

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

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

definition *LowView* **where**
LowView f A x = (*Exists x* (*And* (*View f A* $(\lambda s.\ s\ x)$) (*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-fh h*)) = *f* (*normalize* (*get-fh* h'))
⟨*proof*⟩

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

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:
$(\forall\, v\ v'\ sarg\ sarg'.\ \alpha\ v = \alpha\ v' \wedge spre\ sarg\ sarg' \longrightarrow \alpha\ (sact\ v\ sarg) = \alpha\ (sact\ v'\ sarg')) \wedge$
$(\forall\, v\ v'\ uarg\ uarg'\ i.\ \alpha\ v = \alpha\ v' \wedge upre\ i\ uarg\ uarg' \longrightarrow \alpha\ (uact\ i\ v\ uarg) = \alpha\ (uact\ i\ v'\ uarg')) \wedge$

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

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

3.2 Rules of the Logic

inductive *CommCSL* :: $('i, 'a, nat)\ cont \Rightarrow ('i, 'a, nat)\ assertion \Rightarrow cmd \Rightarrow ('i, 'a, nat)\ assertion \Rightarrow bool$
(‹- ⊢ {-} - {-}› [*51,0,0*] *81*) **where**
$RuleSkip$: $\Delta \vdash \{P\}\ Cskip\ \{P\}$
| $RuleAssign$: $[\![\ \bigwedge\Gamma.\ \Delta = Some\ \Gamma \Longrightarrow x \notin fvA\ (invariant\ \Gamma)\ ;\ collect\text{-}existentials\ P \cap fvE\ E = \{\}\ ]\!] \Longrightarrow \Delta \vdash \{subA\ x\ E\ P\}\ Cassign\ x\ E\ \{P\}$
| $RuleNew$: $[\![\ x \notin fvE\ E;\ \bigwedge\Gamma.\ \Delta = Some\ \Gamma \Longrightarrow x \notin fvA\ (invariant\ \Gamma) \wedge view\text{-}function\text{-}of\text{-}inv\ \Gamma\ ]\!] \Longrightarrow \Delta \vdash \{Emp\}\ Calloc\ x\ E\ \{PointsTo\ (Evar\ x)\ pwrite\ E\}$
| $RuleWrite$: $[\![\ \bigwedge\Gamma.\ \Delta = Some\ \Gamma \Longrightarrow view\text{-}function\text{-}of\text{-}inv\ \Gamma\ ;\ v \notin fvE\ loc\ ]\!]$
$\Longrightarrow \Delta \vdash \{Exists\ v\ (PointsTo\ loc\ pwrite\ (Evar\ v))\}\ Cwrite\ loc\ E\ \{PointsTo\ loc\ pwrite\ E\}$
| $[\![\ \bigwedge\Gamma.\ \Delta = Some\ \Gamma \Longrightarrow x \notin fvA\ (invariant\ \Gamma) \wedge view\text{-}function\text{-}of\text{-}inv\ \Gamma\ ;\ x \notin fvE\ E \cup fvE\ e\ ]\!] \Longrightarrow$
$\Delta \vdash \{PointsTo\ E\ \pi\ e\}\ Cread\ x\ E\ \{And\ (PointsTo\ E\ \pi\ e)\ (Bool\ (Beq\ (Evar\ x)\ e))\}$
| $RuleShare$: $[\![\ \Gamma = (\!|\ view = f,\ abstract\text{-}view = \alpha,\ saction = sact,\ uaction = uact,\ invariant = J\ |\!)\ ;\ all\text{-}axioms\ \alpha\ sact\ spre\ uact\ upre\ ;$
$Some\ \Gamma \vdash \{Star\ P\ EmptyFullGuards\}\ C\ \{Star\ Q\ (And\ (PreSharedGuards\ (Abs\text{-}precondition\ spre))\ (PreUniqueGuards\ (Abs\text{-}indexed\text{-}precondition\ upre)))\};$
$view\text{-}function\text{-}of\text{-}inv\ \Gamma\ ;\ unary\ J\ ;\ precise\ J\ ;\ wf\text{-}indexed\text{-}precondition\ upre\ ;\ wf\text{-}precondition\ spre\ ;\ x \notin fvA\ J\ ;$
$no\text{-}guard\text{-}assertion\ (Star\ P\ (LowView\ (\alpha \circ f)\ J\ x))\ ]\!] \Longrightarrow None \vdash \{Star\ P\ (LowView\ (\alpha \circ f)\ J\ x)\}\ C\ \{Star\ Q\ (LowView\ (\alpha \circ f)\ J\ x)\}$
| $RuleAtomicUnique$: $[\![\ \Gamma = (\!|\ view = f,\ abstract\text{-}view = \alpha,\ saction = sact,\ uaction = uact,\ invariant = J\ |\!)\ ;$
$no\text{-}guard\text{-}assertion\ P\ ;\ no\text{-}guard\text{-}assertion\ Q\ ;$
$None \vdash \{Star\ P\ (View\ f\ J\ (\lambda s.\ s\ x))\}\ C\ \{Star\ Q\ (View\ f\ J\ (\lambda s.\ uact\ index\ (s\ x)\ (map\text{-}to\text{-}arg\ (s\ uarg))))\}\ ;$

precise J ; *unary J* ; *view-function-of-inv* Γ ; $x \notin$ *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-to-list* (*s l*)) ; $x \notin$ *fvS* ($\lambda s.$ *map-to-arg* (*s uarg*) # *map-to-list* (*s l*)) $\rrbracket$
$\Longrightarrow$ *Some* $\Gamma \vdash$ {*Star P* (*UniqueGuard index* ($\lambda s.$ *map-to-list* (*s l*)))} *Catomic C* {*Star Q* (*UniqueGuard index* ($\lambda s.$ *map-to-arg* (*s uarg*) # *map-to-list* (*s l*)))}
| *RuleAtomicShared*: $\llbracket$ Γ = $\llparenthesis$ *view* = *f*, *abstract-view* = α, *saction* = *sact*, *uaction* = *uact*, *invariant* = *J* $\rrparenthesis$; *no-guard-assertion P* ; *no-guard-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-to-arg* (*s sarg*)))))} ;
precise J ; *unary J* ; *view-function-of-inv* Γ ; $x \notin$ *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-to-multiset* (*s ms*)) ; $x \notin$ *fvS* ($\lambda s.$ {# *map-to-arg* (*s sarg*) #} + *map-to-multiset* (*s ms*)) $\rrbracket$
$\Longrightarrow$ *Some* $\Gamma \vdash$ {*Star P* (*SharedGuard* π ($\lambda s.$ *map-to-multiset* (*s ms*)))} *Catomic C* {*Star Q* (*SharedGuard* π ($\lambda s.$ {# *map-to-arg* (*s sarg*) #} + *map-to-multiset* (*s ms*)))}
| *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*) ; $\bigwedge\Gamma.$ Δ = *Some* $\Gamma \Longrightarrow$ *disjoint* (*fvA* (*invariant* Γ)) (*wrC C2*) ;
$\bigwedge\Gamma.$ Δ = *Some* $\Gamma \Longrightarrow$ *disjoint* (*fvA* (*invariant* Γ)) (*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* ; $\bigwedge\Gamma.$ Δ = *Some* $\Gamma \Longrightarrow x \notin$ *fvA* (*invariant* Γ) ; *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 $\alpha\ v = \alpha\ v' \wedge spre\ sarg\ sarg' \Longrightarrow \alpha\ (sact\ v\ sarg) = \alpha\ (sact\ v'\ sarg')$
$\langle proof \rangle$

lemma *uabstract*:
assumes *all-axioms* α *sact spre uact upre*
shows $\alpha\ v = \alpha\ v' \wedge upre\ i\ uarg\ uarg' \Longrightarrow \alpha\ (uact\ i\ v\ uarg) = \alpha\ (uact\ i\ v'\ uarg')$
$\langle proof \rangle$

lemma *spre-refl*:
assumes *all-axioms* α *sact spre uact upre*
shows $spre\ sarg\ sarg' \Longrightarrow spre\ sarg'\ sarg'$
$\langle proof \rangle$

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

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

lemma *su-com*:
assumes *all-axioms* α *sact spre uact upre*
shows $\alpha\ v = \alpha\ v' \Longrightarrow spre\ sarg\ sarg \wedge upre\ i\ uarg\ uarg \Longrightarrow \alpha\ (sact\ (uact\ i\ v\ uarg)\ sarg) = \alpha\ (uact\ i\ (sact\ v'\ sarg)\ uarg)$
$\langle proof \rangle$

lemma *uu-com*:
assumes *all-axioms* α *sact spre uact upre*
and $i \neq i'$

and $\alpha\ v = \alpha\ v'$
and $upre\ i'\ uarg'\ uarg'$
and $upre\ i\ uarg\ uarg$
shows $\alpha\ (uact\ i'\ (uact\ i\ v\ uarg)\ uarg') = \alpha\ (uact\ i\ (uact\ i'\ v'\ uarg')\ uarg)$
$\langle proof \rangle$

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'*
$\langle proof \rangle$

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*
$\langle proof \rangle$

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*)
$\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *PRE-unique-tlI*:
assumes *PRE-unique upre qa qb*
and *upre ta tb*
shows *PRE-unique upre* (*ta* # *qa*) (*tb* # *qb*)
⟨*proof*⟩

fun *abstract-pre* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('i \Rightarrow 'b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('i, 'a, 'b)\ action$ $\Rightarrow ('i, 'a, 'b)\ action \Rightarrow bool$ **where**
abstract-pre spre upre (*Shared sarg*) (*Shared sarg'*) ⟷ *spre sarg sarg'*
| *abstract-pre spre upre* (*Unique k uarg*) (*Unique k' uarg'*) ⟷ $k = k' \wedge$ *upre k uarg uarg'*
| *abstract-pre spre upre* - - ⟷ *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' ⟷ $length\ s = length\ s' \wedge (\forall\, i.\ i \geq 0 \wedge i < length\ s \longrightarrow abstract\text{-}pre\ spre\ upre\ (s\ !\ i)\ (s'\ !\ i))$

lemma *PRE-sequenceE*:
assumes *PRE-sequence spre upre s s'*
and $i \geq 0 \wedge i < length\ s$
shows *abstract-pre spre upre* (*s* ! *i*) (*s'* ! *i*)
⟨*proof*⟩

lemma *PRE-sequenceI*:
  assumes $length\ s = length\ s'$
    and $\bigwedge i.\ i \geq 0 \wedge i < length\ s \Longrightarrow abstract\text{-}pre\ spre\ upre\ (s\ !\ i)\ (s'\ !\ i)$
  shows *PRE-sequence spre upre s s'*
  ⟨*proof*⟩

lemma *PRE-sequenceI-rec*:
  assumes *PRE-sequence spre upre s s'*
    and *abstract-pre spre upre a b*
  shows *PRE-sequence spre upre* $(a\ \#\ s)\ (b\ \#\ s')$
  ⟨*proof*⟩

lemma *PRE-sequenceE-rec*:
  assumes *PRE-sequence spre upre* $(a\ \#\ s)\ (b\ \#\ s')$
  shows *PRE-sequence spre upre s s'*
    and *abstract-pre spre upre a b*
  ⟨*proof*⟩

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

lemma *obtain-other-elem-ms*:
  assumes *PRE-shared spre sargs sargs'*
    and $sarg \in\#\ sargs$
  shows $\exists\, sarg'.\ sarg' \in\#\ sargs' \wedge spre\ sarg\ sarg' \wedge PRE\text{-}shared\ spre\ (sargs - \{\#\ sarg\ \#\})\ (sargs' - \{\#\ sarg'\ \#\})$
⟨*proof*⟩

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

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

  shows $\exists\, s''.\ possible\text{-}sequence\ sargs'\ uargs'\ s'' \wedge PRE\text{-}sequence\ spre\ upre\ s\ s''$
  ⟨*proof*⟩

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*⟩

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

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

lemma *abstract-pre-refl*:
assumes $abstract\text{-}pre\ spre\ upre\ a\ b$
and $all\text{-}axioms\ \alpha\ sact\ spre\ uact\ upre$
shows $abstract\text{-}pre\ spre\ upre\ b\ b$
⟨*proof*⟩

lemma *PRE-sequence-refl*:
assumes $PRE\text{-}sequence\ spre\ upre\ s\ s'$
and $all\text{-}axioms\ \alpha\ sact\ spre\ uact\ upre$
shows $PRE\text{-}sequence\ spre\ upre\ s'\ s'$
⟨*proof*⟩

lemma *PRE-sequence-removes*:
assumes $PRE\text{-}sequence\ spre\ upre\ s\ s$
shows $PRE\text{-}sequence\ spre\ upre\ (remove\text{-}at\text{-}index\ n\ s)\ (remove\text{-}at\text{-}index\ n\ s)$
⟨*proof*⟩

lemma *PRE-sequence-insert*:
assumes $abstract\text{-}pre\ spre\ upre\ x\ x$
and $PRE\text{-}sequence\ spre\ upre\ s\ s$
shows $PRE\text{-}sequence\ spre\ upre\ (insert\text{-}at\ n\ x\ s)\ (insert\text{-}at\ n\ x\ s)$
⟨*proof*⟩

lemma *empty-possible-sequence*:
assumes $possible\text{-}sequence\ sargs\ uargs\ []$
and $possible\text{-}sequence\ sargs\ uargs\ s'$
shows $s' = []$
⟨*proof*⟩

lemma *it-all-commutes*:
assumes $possible\text{-}sequence\ sargs\ uargs\ s$

and *possible-sequence sargs uargs s'*
and α *v0* = α *v0'*
and *PRE-sequence spre upre s s*
and *PRE-sequence spre upre s' s'*
and *all-axioms* α *sact spre uact upre*
shows α (*compute sact uact v0 s*) = α (*compute sact uact v0' s'*)
⟨*proof*⟩

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

lemma *simple-possible-PRE-seq*:
assumes *possible-sequence sargs uargs s*
and *possible-sequence sargs' uargs' s'*
and *PRE-shared spre sargs sargs'*
and $\bigwedge k$. *PRE-unique* (*upre k*) (*uargs k*) (*uargs' k*)
and *all-axioms* α *sact spre uact upre*
shows *PRE-sequence spre upre s' s'*
⟨*proof*⟩

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

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

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

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

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

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

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

assumes *reachable-value sact uact v0 sargs uargs v1*
shows $\exists$ *s. possible-sequence sargs uargs s* $\wedge$ *v1* = *compute sact uact v0 s*
$\langle proof \rangle$

lemma *PRE-shared-simpler-implies*:
assumes *PRE-shared-simpler spre a b*
shows *PRE-shared spre a b*
$\langle proof \rangle$

The following theorem corresponds to Lemma 4.2 in the paper.

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

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

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

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

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

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

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

lemma *semi-consistentE*:
  assumes *semi-consistent* Γ *v0 h*
  shows ∃ *sargs uargs*. *get-gs h* = *Some* (*pwrite*, *sargs*) ∧ (∀ *k*. *get-gu h k* = *Some* (*uargs k*))
   ∧ *reachable-value* (*saction* Γ) (*uaction* Γ) *v0 sargs uargs* (*view* Γ (*normalize* (*get-fh h*)))
⟨*proof*⟩

lemma *semi-consistentI*:
  assumes *all-guards h*
    and *reachable* Γ *v0 h*
   shows *semi-consistent* Γ *v0 h*
  ⟨*proof*⟩

lemma *no-guard-then-smaller-same*:
  assumes *Some h* = *Some a* ⊕ *Some b*
    and *no-guard h*
   shows *no-guard a*
⟨*proof*⟩

lemma *all-guardsI*:
  assumes $\bigwedge k$. *get-gu h k* ≠ *None*
    and ∃ *v*. *get-gs h* = *Some* (*pwrite*, *v*)
   shows *all-guards h*
  ⟨*proof*⟩

lemma *all-guards-same*:
  assumes *all-guards a*

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

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

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

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

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

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

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

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

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

definition *pair-sat* :: (*store* $\times$ $('i, 'a)$ *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

$\langle proof \rangle$

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

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

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* $SA'\ SB'$) ($Star\ A\ B$)
$\langle proof \rangle$

lemma *add-states-subset*:
assumes $S1 \subseteq S1'$
shows *add-states* $S1\ S2 \subseteq$ *add-states* $S1'\ S2$
$\langle proof \rangle$

lemma *add-states-comm*:
add-states $S1\ S2 =$ *add-states* $S2\ S1$
$\langle proof \rangle$

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

lemma *magic-lemma*:
assumes $Some\ x1 = Some\ a1 \oplus Some\ j1$
and $Some\ x2 = Some\ a2 \oplus Some\ j2$
and $(s1,\ x1),\ (s2,\ x2) \models Star\ A\ J$
and $(s1,\ j1),\ (s2,\ j2) \models J$
and *precise* J
shows $(s1,\ a1),\ (s2,\ a2) \models A$
$\langle proof \rangle$

lemma *full-no-guard-same-normalize*:
assumes *full-ownership* (*get-fh* h) $\wedge$ *no-guard* h
and *full-ownership* (*get-fh* h') $\wedge$ *no-guard* h'
and *normalize* (*get-fh* h) $=$ *normalize* (*get-fh* h')

shows $h = h'$
$\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

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

lemma *no-guards-remove*:
no-guard h $\longleftrightarrow$ h = *remove-guards h*
⟨*proof*⟩

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

lemma *get-fh-add-sguard*:
get-fh (*add-sguard-to-no-guard h* π *ms*) = *get-fh h*
⟨*proof*⟩

lemma *add-sguard-as-sum*:
assumes *no-guard h*
shows *Some* (*add-sguard-to-no-guard h* π *ms*) = *Some h* $\oplus$ *Some* (*Map.empty*, *Some* (π, *ms*), (λ-. *None*))
⟨*proof*⟩

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

lemma *get-fh-add-uguard*:
get-fh (*add-uguard-to-no-guard k h l*) = *get-fh h*
⟨*proof*⟩

lemma *prove-sum*:
assumes a ## b
and $\bigwedge x$. *Some x* = *Some a* $\oplus$ *Some b* $\Longrightarrow$ $x = y$
shows *Some y* = *Some a* $\oplus$ *Some b*
⟨*proof*⟩

lemma *add-uguard-as-sum*:
assumes *no-guard h*
shows *Some* (*add-uguard-to-no-guard k h l*) = *Some h* $\oplus$ *Some* (*Map.empty*, *None*, (λ-. *None*)(k := *Some l*))
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *denormalize-properties*:
shows $no\text{-}guard\ (denormalize\ H)$
and $full\text{-}ownership\ (get\text{-}fh\ (denormalize\ H))$
and $normalize\ (get\text{-}fh\ (denormalize\ H)) = H$
and $full\text{-}ownership\ (get\text{-}fh\ h) \land no\text{-}guard\ h \Longrightarrow denormalize\ (normalize\ (get\text{-}fh$

h)) = *h*
and *full-ownership* (*get-fh* *h*) $\Longrightarrow$ *denormalize* (*normalize* (*get-fh* *h*)) = *remove-guards* *h*
⟨*proof*⟩

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

lemma *no-guard-then-sat-star*:
assumes *no-guard* *h* ∧ *no-guard* *h'*
and (*s*, *h*), (*s'*, *h'*) ⊨ *Q*
shows (*s*, *add-sguard-to-no-guard* *h* π (*ms* *s*)), (*s'*, *add-sguard-to-no-guard* *h'* π (*ms* *s'*)) ⊨ *Star* *Q* (*SharedGuard* π *ms*)
⟨*proof*⟩

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
⟨*proof*⟩

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

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

Only need to agree on vars

definition *upperize* **where**
  $upperize\ S\ vars = \{\ \sigma'\ |\sigma\ \sigma'.\ \sigma \in S \wedge snd\ \sigma = snd\ \sigma' \wedge agrees\ vars\ (fst\ \sigma)\ (fst\ \sigma')\}$

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

lemma *upper-fvsI*:
  assumes $\bigwedge s\ s'\ h.\ (s,\ h) \in S \wedge agrees\ vars\ s\ s' \Longrightarrow (s',\ h) \in S$
  shows $upper\text{-}fvs\ S\ vars$
  ⟨*proof*⟩

lemma *pair-sat-comm*:
  assumes $pair\text{-}sat\ S\ S'\ A$
  shows $pair\text{-}sat\ S'\ S\ A$
⟨*proof*⟩

lemma *in-upperize*:
  $(s',\ h) \in upperize\ S\ vars \longleftrightarrow (\exists\, s.\ (s,\ h) \in S \wedge agrees\ vars\ s\ s')$ (**is** $?A \longleftrightarrow\ ?B$)
⟨*proof*⟩

lemma *upper-fvs-upperize*:
  $upper\text{-}fvs\ (upperize\ S\ vars)\ vars$
⟨*proof*⟩

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

lemma *pair-sat-upperize*:
  assumes $pair\text{-}sat\ S\ S'\ A$
  shows $pair\text{-}sat\ (upperize\ S\ (fvA\ A))\ S'\ A$
⟨*proof*⟩

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

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

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

$\langle proof \rangle$

lemma *close-var-subset*:
$S \subseteq$ *close-var* S x
$\langle proof \rangle$

lemma *upper-fvs-close-vars*:
upper-fvs (*close-var* S x) $(-\ \{x\})$
$\langle proof \rangle$

lemma *sat-inv-agrees*:
assumes *sat-inv* s *hj* Γ
and *agrees* (*fvA* (*invariant* Γ)) s s'
shows *sat-inv* s' *hj* Γ
$\langle proof \rangle$

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

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'))
$\langle proof \rangle$

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* $hf \wedge \Delta =$ *None* $\wedge$ *full-ownership* (*get-fh* H) $\wedge$ *no-guard* $H \Longrightarrow \neg$ *aborts* C $(s,$ *normalize* (*get-fh* $H))$
and $\bigwedge H\ hf\ hj\ v0\ \Gamma.\ \Delta =$ *Some* $\Gamma \wedge$ *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*$)$
$\langle proof \rangle$

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

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

lemma *no-abortE*:
assumes $no\text{-}abort\ \Delta\ C\ s\ h$
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 $\Delta = Some\ \Gamma \Longrightarrow Some\ H = Some\ h \oplus Some\ hj \oplus Some\ hf \Longrightarrow sat\text{-}inv\ s\ hj\ \Gamma \Longrightarrow full\text{-}ownership\ (get\text{-}fh\ H) \Longrightarrow semi\text{-}consistent\ \Gamma\ v0\ H$
$\Longrightarrow \neg\ aborts\ C\ (s,\ normalize\ (get\text{-}fh\ H))$
$\langle proof \rangle$

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

fun $safe :: nat \Rightarrow ('i,\ 'a,\ nat)\ cont \Rightarrow cmd \Rightarrow (store \times ('i,\ 'a)\ heap) \Rightarrow (store \times ('i,\ 'a)\ heap)\ set \Rightarrow bool$ **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-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 $safe\ (Suc\ n)\ (None :: ('i,\ 'a,\ nat)\ cont)\ C\ (s,\ h :: ('i,\ 'a)\ heap)\ S$
⟨*proof*⟩

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

lemma *safeI*:
fixes $\Delta :: ('i,\ 'a,\ nat)\ cont$
assumes $C = Cskip \Longrightarrow (s, h) \in S$
and *no-abort* Δ *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*⟩

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

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

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

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

lemma *safeNoneE-bis*:

fixes *no-cont* :: ('*i*, '*a*, *nat*) *cont*
assumes *safe* (*Suc n*) *no-cont C* (*s*, *h* :: ('*i*, '*a*) *heap*) *S*
and *no-cont* = *None*
shows $C = Cskip \Longrightarrow (s, h) \in S$
and *no-abort no-cont C s h*
and *Some H* = *Some h* ⊕ *Some hf* ⟹ *full-ownership* (*get-fh H*) ⟹ *no-guard H* ⟹ *red C* (*s*, *normalize* (*get-fh H*)) *C'* (*s'*, *h'*)
⟹ (∃ *h'' H'*. *full-ownership* (*get-fh H'*) ∧ *no-guard H'* ∧ *h'* = *normalize* (*get-fh H'*) ∧ *Some H'* = *Some h''* ⊕ *Some hf* ∧ *safe n no-cont C'* (*s'*, *h''*) *S*)
⟨*proof*⟩

4.3.3 Useful results about safety

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

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'*
⟨*proof*⟩

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

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

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

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* $(fvA\ (invariant\ \Gamma))$ $s0$ $s1$
shows *safe* n Δ C $(s1, h)$ S
⟨*proof*⟩

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

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

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

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

4.3.4 Hoare triples

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

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

lemma *hoare-triple-validI*:
assumes $\bigwedge s\ h\ n.\ (s,\ h),\ (s,\ h) \models P \Longrightarrow \mathit{safe}\ n\ \Gamma\ C\ (s,\ h)\ (\Sigma\ (s,\ h))$
and $\bigwedge s\ h\ s'\ h'.\ (s,\ h),\ (s',\ h') \models P \Longrightarrow \mathit{pair\text{-}sat}\ (\Sigma\ (s,\ h))\ (\Sigma\ (s',\ h'))\ Q$
shows *hoare-triple-valid* $\Gamma\ P\ C\ Q$
⟨*proof*⟩

lemma *hoare-triple-validI-bounded*:
assumes $\bigwedge s\ h\ n.\ (s,\ h),\ (s,\ h) \models P \Longrightarrow \mathit{bounded}\ h \Longrightarrow \mathit{safe}\ n\ \Gamma\ C\ (s,\ h)\ (\Sigma\ (s,\ h))$
and $\bigwedge s\ h\ s'\ h'.\ (s,\ h),\ (s',\ h') \models P \Longrightarrow \mathit{pair\text{-}sat}\ (\Sigma\ (s,\ h))\ (\Sigma\ (s',\ h'))\ Q$
shows *hoare-triple-valid* $\Gamma\ P\ C\ Q$
⟨*proof*⟩

lemma *hoare-triple-valid-smallerI*:
assumes $\bigwedge \sigma\ n.\ \sigma,\ \sigma \models P \Longrightarrow \mathit{safe}\ n\ \Gamma\ C\ \sigma\ (\Sigma\ \sigma)$
and $\bigwedge \sigma\ \sigma'.\ \sigma,\ \sigma' \models P \Longrightarrow \mathit{pair\text{-}sat}\ (\Sigma\ \sigma)\ (\Sigma\ \sigma')\ Q$
shows *hoare-triple-valid* $\Gamma\ P\ C\ Q$
⟨*proof*⟩

lemma *hoare-triple-valid-smallerI-bounded*:
assumes $\bigwedge \sigma\ n.\ \sigma,\ \sigma \models P \Longrightarrow \mathit{bounded}\ (\mathit{snd}\ \sigma) \Longrightarrow \mathit{safe}\ n\ \Gamma\ C\ \sigma\ (\Sigma\ \sigma)$
and $\bigwedge \sigma\ \sigma'.\ \sigma,\ \sigma' \models P \Longrightarrow \mathit{pair\text{-}sat}\ (\Sigma\ \sigma)\ (\Sigma\ \sigma')\ Q$
shows *hoare-triple-valid* $\Gamma\ P\ C\ Q$
⟨*proof*⟩

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

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

end

4.4 Soundness of the Rules

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

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

theory *Soundness*
  imports *Safety AbstractCommutativity*
begin

4.4.1 Skip

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

theorem *rule-skip*:
  hoare-triple-valid Γ *P Cskip P*
$\langle proof \rangle$

4.4.2 Assign

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

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

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

4.4.3 Alloc

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

lemma *safe-new-None*:

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

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

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

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

4.4.4 Write

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

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

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

lemma *safe-write*:

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

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

4.4.5 Read

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

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

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

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

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

4.4.6 Share

lemma *share-no-abort*:

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

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

lemma *share-lemma*:
assumes *safe n* (*Some* Γ) *C* (*s*, *h* :: (′*i*, ′*a*) *heap*) *S*
and *Some* (*h*′ :: (′*i*, ′*a*) *heap*) = *Some h* ⊕ *Some hj*
and *sat-inv s hj* Γ
and *semi-consistent* Γ *v0 h*′
and *view-function-of-inv* Γ
and *bounded h*′
shows *safe n* (*None* :: (′*i*, ′*a*, *nat*) *cont*) *C* (*s*, *remove-guards h*′) (*S-after-share S* Γ *v0*)
⟨*proof*⟩

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

lemma *has-guard-then-safe-none*:
assumes ¬ *no-guard h*
and *C* = *Cskip* ⟹ (*s*, *h*) ∈ *S*
and *accesses C s* ⊆ *dom* (*fst h*) ∧ *writes C s* ⊆ *fpdom* (*fst h*)
shows *safe n* (*None* :: (′*i*, ′*a*, *nat*) *cont*) *C* (*s*, *h*) *S*
⟨*proof*⟩

theorem *share-rule*:
fixes Γ :: (′*i*, ′*a*, *nat*) *single-context*
assumes Γ = (| *view* = *f*, *abstract-view* = α, *saction* = *sact*, *uaction* = *uact*, *invariant* = *J* |)
and *all-axioms* α *sact spre uact upre*
and *hoare-triple-valid* (*Some* Γ) (*Star P EmptyFullGuards*) *C* (*Star Q* (*And* (*PreSharedGuards* (*Abs-precondition spre*)) (*PreUniqueGuards* (*Abs-indexed-precondition upre*))))
and *view-function-of-inv* Γ

and *unary J* $\wedge$ *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*⟩

4.4.7 Atomic

lemma *red-rtrans-induct*:
assumes *red-rtrans C* σ C' σ'
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 $P\ C\ \sigma\ C'\ \sigma'$
⟨*proof*⟩

lemma *safe-atomic*:
assumes *red-rtrans C1* $\sigma 1$ *C2* $\sigma 2$
and $\sigma 1 = (s1, H1)$
and $\sigma 2 = (s2, H2)$
and $\bigwedge n.$ *safe n* (*None* :: ('*i*, '*a*, *nat*) *cont*) *C1* (*s1*, *h*) *S*
and *H* = *denormalize H1*
and *Some H* = *Some h* $\oplus$ *Some hf*
and *full-ownership* (*get-fh H*) $\wedge$ *no-guard H*
shows $\neg$ *aborts C2* $\sigma 2$ $\wedge$ (*C2* = *Cskip* $\longrightarrow$
($\exists$ *h1* H'. *Some* H' = *Some h1* $\oplus$ *Some hf* $\wedge$ *H2* = *normalize* (*get-fh* (H')) $\wedge$ *no-guard* H' $\wedge$ *full-ownership* (*get-fh* H') $\wedge$ (*s2*, *h1*) $\in$ *S*))
⟨*proof*⟩

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

fixes *map-to-list* :: *nat* $\Rightarrow$ '*a list*
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* ($\lambda s.\ s\ x$)))
C (*Star Q* (*View f J* ($\lambda s.$ *uact index* (*s x*) (*map-to-arg* (*s uarg*))))))

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 *uarg* $\notin$ *fvC C*
and $l \notin$ *fvC C*

and $x \notin fvS$ ($\lambda s.$ *map-to-list* (s l))
and $x \notin fvS$ ($\lambda s.$ *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* ($\lambda s.$ *map-to-list* (s l)))) (*Catomic* C)
(*Star* Q (*UniqueGuard index* ($\lambda s.$ *map-to-arg* (s *uarg*) # *map-to-list* (s l))))
$\langle proof \rangle$

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

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

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

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

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

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 $C1'$ σ' $\Longrightarrow$ P
and $\bigwedge C2'.$ C' = *Cpar* $C1$ $C2'$ $\wedge$ *red* $C2$ σ $C2'$ σ' $\Longrightarrow$ P

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

lemma *no-abort-par*:
assumes $no\text{-}abort\ \Gamma\ C1\ s\ h$
and $no\text{-}abort\ \Gamma\ C2\ s\ h$
and $safe\ (Suc\ n)\ \Delta\ C1\ (s,\ h1)\ S1$
and $safe\ (Suc\ n)\ \Delta\ C2\ (s,\ h2)\ S2$
and $Some\ h = Some\ h1 \oplus Some\ h2$
and $bounded\ h$
shows $no\text{-}abort\ \Gamma\ (Cpar\ C1\ C2)\ s\ h \wedge accesses\ (Cpar\ C1\ C2)\ s \subseteq dom\ (fst\ h) \wedge writes\ (Cpar\ C1\ C2)\ s \subseteq fpdom\ (fst\ h)$
⟨*proof*⟩

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

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

and $upper\text{-}fvs\ S1\ vars1$
and $upper\text{-}fvs\ S2\ vars2$

and $bounded\ h$

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

lemma *parallel-comp-some*:
assumes $safe\ n\ (Some\ \Gamma)\ C1\ (s,\ h1)\ S1$
and $safe\ n\ (Some\ \Gamma)\ C2\ (s,\ h2)\ S2$
and $Some\ h = Some\ h1 \oplus Some\ h2$

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

and $upper\text{-}fvs\ S1\ vars1$
and $upper\text{-}fvs\ S2\ vars2$

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

and $bounded\ h$

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

lemma *parallel-comp*:
fixes 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* $\oplus$ *Some h2*
and *disjoint* (*fvC C1* $\cup$ *vars1*) (*wrC C2*)
and *disjoint* (*fvC C2* $\cup$ *vars2*) (*wrC C1*)
and *upper-fvs S1 vars1*
and *upper-fvs S2 vars2*

and $\bigwedge\Gamma$. Δ = *Some* Γ $\Longrightarrow$ *disjoint* (*fvA* (*invariant* Γ)) (*wrC C2*)
and $\bigwedge\Gamma$. Δ = *Some* Γ $\Longrightarrow$ *disjoint* (*fvA* (*invariant* Γ)) (*wrC C1*)

and *bounded h*

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

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

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

and $\bigwedge\Gamma$. Δ = *Some* Γ $\Longrightarrow$ *disjoint* (*fvA* (*invariant* Γ)) (*wrC C2*)
and $\bigwedge\Gamma$. Δ = *Some* Γ $\Longrightarrow$ *disjoint* (*fvA* (*invariant* Γ)) (*wrC C1*)

and *precise P1* $\vee$ *precise P2*

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

4.4.9 If

lemma *if-cases*:
assumes *red* (*Cif b C1 C2*) (*s*, *h*) *C'* (*s'*, *h'*)
and *C'* = *C1* $\Longrightarrow$ *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*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

theorem *if1-rule*:
fixes Δ :: (*'i*, *'a*, *nat*) *cont*
assumes *hoare-triple-valid* Δ (*And P* (*Bool b*)) *C1 Q*
and *hoare-triple-valid* Δ (*And P* (*Bool* (*Bnot b*))) *C2 Q*
shows *hoare-triple-valid* Δ (*And P* (*Low b*)) (*Cif b C1 C2*) *Q*
⟨*proof*⟩

theorem *if2-rule*:
fixes Δ :: (*'i*, *'a*, *nat*) *cont*
assumes *hoare-triple-valid* Δ (*And P* (*Bool b*)) *C1 Q*
and *hoare-triple-valid* Δ (*And P* (*Bool* (*Bnot b*))) *C2 Q*
and *unary Q*
shows *hoare-triple-valid* Δ *P* (*Cif b C1 C2*) *Q*
⟨*proof*⟩

4.4.10 Sequential composition

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

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

lemma *safe-seq-None*:

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

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

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

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

4.4.11 Frame rule

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

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

lemma *safe-frame*:
fixes $\Delta :: ('i, 'a, nat)\ cont$
assumes $safe\ n\ \Delta\ C\ (s, h)\ S$
and $Some\ H = Some\ h \oplus Some\ hf0$
and $bounded\ H$

shows *safe* $n\ \Delta\ C\ (s,\ H)\ (\mathit{add\text{-}states}\ S\ \{(s'',\ \mathit{hf0})\ |s''.\ \mathit{agrees}\ (-\ \mathit{wrC}\ C)\ s\ s''\})$
⟨*proof*⟩

theorem *frame-rule*:
fixes $\Delta :: ('i,\ 'a,\ nat)\ cont$
assumes *hoare-triple-valid* $\Delta\ P\ C\ Q$
and *disjoint* $(\mathit{fvA}\ R)\ (\mathit{wrC}\ C)$
and *precise* $P \lor$ *precise* R
shows *hoare-triple-valid* $\Delta\ (\mathit{Star}\ P\ R)\ C\ (\mathit{Star}\ Q\ R)$
⟨*proof*⟩

4.4.12 Consequence

theorem *consequence-rule*:
fixes $\Delta :: ('i,\ 'a,\ nat)\ cont$
assumes *hoare-triple-valid* $\Delta\ P'\ C\ Q'$
and *entails* $P\ P'$
and *entails* $Q'\ Q$
shows *hoare-triple-valid* $\Delta\ P\ C\ Q$
⟨*proof*⟩

4.4.13 Existential

theorem *existential-rule*:
fixes $\Delta :: ('i,\ 'a,\ nat)\ cont$
assumes *hoare-triple-valid* $\Delta\ P\ C\ Q$
and $x \notin \mathit{fvC}\ C$
and $\bigwedge\Gamma.\ \Delta = \mathit{Some}\ \Gamma \Longrightarrow x \notin \mathit{fvA}\ (\mathit{invariant}\ \Gamma)$
and *unambiguous* $P\ x$
shows *hoare-triple-valid* $\Delta\ (\mathit{Exists}\ x\ P)\ C\ (\mathit{Exists}\ x\ Q)$
⟨*proof*⟩

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* $b\ (\mathit{fst}\ \sigma')$; $\sigma'' \in \Sigma\ \sigma'\ ]\!] \Longrightarrow$ *leads-to-loop* $b\ I\ \Sigma\ \sigma\ \sigma''$

definition *leads-to-loop-set* **where**
leads-to-loop-set $b\ I\ \Sigma\ \sigma = \{\ \sigma'\ |\sigma'.$ *leads-to-loop* $b\ I\ \Sigma\ \sigma\ \sigma'\}$

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

inductive-cases *red-while-cases*: *red* $(\mathit{Cwhile}\ b\ s)\ \sigma\ C'\ \sigma'$
inductive-cases *abort-while-cases*: *aborts* $(\mathit{Cwhile}\ b\ s)\ \sigma$

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* σ ($\Sigma\ \sigma$)
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* $\Sigma\ \sigma$)
⟨*proof*⟩

lemma *safe-while-Some*:
assumes $\bigwedge\sigma\ m.\ \sigma,\ \sigma \models$ *And I* (*Bool b*) $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe n* (*Some* Γ) *C* σ ($\Sigma\ \sigma$)
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)
shows *safe n* (*Some* Γ) (*Cwhile b C*) (s, h) (*trans*-Σ *b I* $\Sigma\ \sigma$)
⟨*proof*⟩

lemma *safe-while*:
fixes Δ :: ('*i*, '*a*, *nat*) *cont*
assumes $\bigwedge\sigma\ m.\ \sigma,\ \sigma \models$ *And I* (*Bool b*) $\Longrightarrow$ *bounded* (*snd* σ) $\Longrightarrow$ *safe n* Δ *C* σ ($\Sigma\ \sigma$)
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* Δ (*Cwhile b C*) (s, h) (*trans*-Σ *b I* $\Sigma\ \sigma$)
⟨*proof*⟩

lemma *leads-to-sat-inv-unary*:
assumes *leads-to-loop b I* $\Sigma\ \sigma\ \sigma'$
and $\bigwedge\sigma\ \sigma'.\ \sigma,\ \sigma' \models$ (*And I* (*Bool b*)) $\Longrightarrow$ *pair-sat* ($\Sigma\ \sigma$) ($\Sigma\ \sigma'$) *I*
and $\sigma, \sigma \models I$
shows $\sigma', \sigma' \models I$
⟨*proof*⟩

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

fun *iterate-sigma* :: *nat* $\Rightarrow$ *bexp* $\Rightarrow$ ('*i*, '*a*, *nat*) *assertion* $\Rightarrow$ ((*store* $\times$ ('*i*, '*a*) *heap*) $\Rightarrow$ (*store* $\times$ ('*i*, '*a*) *heap*) *set*) $\Rightarrow$ (*store* $\times$ ('*i*, '*a*) *heap*) $\Rightarrow$ (*store* $\times$ ('*i*, '*a*) *heap*) *set*
where
iterate-sigma 0 b I $\Sigma\ \sigma = \{\sigma\}$

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

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

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*⟩

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*))
⟨*proof*⟩

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 = {}
⟨*proof*⟩

lemma *all-same*:
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* $\sigma 2$)
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*⟩

lemma *non-empty-at-most-once*:
assumes $\bigwedge\sigma\ \sigma'$. σ, $\sigma' \models$ (*And I* (*Bool b*)) $\Longrightarrow$ *pair-sat* (Σ σ) (Σ σ') (*And I* (*Low b*))
and σ, $\sigma \models I$
and *Set.filter* ($\lambda\sigma$. $\neg$ *bdenot b* (*fst* σ)) (*iterate-sigma n1 b I* Σ σ) $\neq$ {}
and *Set.filter* ($\lambda\sigma$. $\neg$ *bdenot b* (*fst* σ)) (*iterate-sigma n2 b I* Σ σ) $\neq$ {}
shows *n1* = *n2*
⟨*proof*⟩

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

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

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

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

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

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

e') C) ($Cwhile$ e C)) (And ($Star$ P' R) ($Bool$ ($Bnot$ e)))
⟨*proof*⟩

4.4.15 CommCSL is sound

theorem *soundness*:
  assumes $\Delta \vdash \{P\}\ C\ \{Q\}$
  shows $\Delta \models \{P\}\ C\ \{Q\}$
  ⟨*proof*⟩

4.5 Corollaries

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

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

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

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

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

  shows $\bigwedge \sigma'\ C'$. *red-rtrans* C ($s1$, *normalize* (*get-fh* $H1$)) C' σ' $\Longrightarrow \neg$ *aborts* C' σ'
    and $\bigwedge \sigma'\ C'$. *red-rtrans* C ($s2$, *normalize* (*get-fh* $H2$)) C' σ' $\Longrightarrow \neg$ *aborts* C' σ'
    and $\bigwedge \sigma 1'\ \sigma 2'$. *red-rtrans* C ($s1$, *normalize* (*get-fh* $H1$)) *Cskip* $\sigma 1'$
  $\Longrightarrow$ *red-rtrans* C ($s2$, *normalize* (*get-fh* $H2$)) *Cskip* $\sigma 2'$
  $\Longrightarrow$ ($\exists\, h1'\ h2'\ H1'\ H2'$. *no-guard* $H1' \land$ *full-ownership* (*get-fh* $H1'$) $\land$ *snd* $\sigma 1'$ = *normalize* (*get-fh* $H1'$) $\land$ $Some\ H1' = Some\ h1' \oplus Some\ hf1$
    $\land$ *no-guard* $H2' \land$ *full-ownership* (*get-fh* $H2'$) $\land$ *snd* $\sigma 2'$ = *normalize* (*get-fh* $H2'$) $\land$ $Some\ H2' = Some\ h2' \oplus Some\ hf2$
    $\land$ (*fst* $\sigma 1'$, $h1'$), (*fst* $\sigma 2'$, $h2'$) $\models Q$)
⟨*proof*⟩

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

This second corollary considers that the heap h1 is the only execution that matters, and thus it ignores any frame. It corresponds to Corollary 4.5 in the paper.

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

    and *full-ownership* $(get\text{-}fh\ H1) \wedge no\text{-}guard\ H1$
    and *full-ownership* $(get\text{-}fh\ H2) \wedge no\text{-}guard\ H2$

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

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**
  $low\text{-}list\ [] = Bool\ Btrue$
| $low\text{-}list\ (v\ \#\ q) = And\ (LowExp\ (Evar\ v))\ (low\text{-}list\ q)$

lemma *low-listE*:
  assumes $(s1, h1), (s2, h2) \models low\text{-}list\ l$
    and $x \in set\ l$
   shows $s1\ x = s2\ x$
  ⟨*proof*⟩

lemma *low-listI*:
  assumes $\bigwedge x.\ x \in set\ l \Longrightarrow s1\ x = s2\ x$
  shows $(s1, h1), (s2, h2) \models low\text{-}list\ l$
  ⟨*proof*⟩

corollary *non-interference*:
  assumes $(None :: ('i, 'a, nat)\ cont) \vdash \{And\ P\ (low\text{-}list\ In)\}\ C\ \{low\text{-}list\ Out\}$
    and $red\text{-}rtrans\ C\ (s1, normalize\ (get\text{-}fh\ H1))\ Cskip\ (s1', h1')$
    and $red\text{-}rtrans\ C\ (s2, normalize\ (get\text{-}fh\ H2))\ Cskip\ (s2', h2')$
    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 $full\text{-}ownership\ (get\text{-}fh\ H1) \wedge no\text{-}guard\ H1$
and $full\text{-}ownership\ (get\text{-}fh\ H2) \wedge no\text{-}guard\ H2$
shows $s1'\ x = s2'\ x$
$\langle proof \rangle$

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

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

corollary *non-interference-no-precondition*:
assumes $(None :: ('i, 'a, nat)\ cont) \vdash \{low\text{-}list\ In\}\ C\ \{low\text{-}list\ Out\}$
and $red\text{-}rtrans\ C\ (s1, h1)\ Cskip\ (s1', h1')$
and $red\text{-}rtrans\ C\ (s2, h2)\ Cskip\ (s2', h2')$
and $\bigwedge x.\ x \in set\ In \Longrightarrow s1\ x = s2\ x$
and $x \in set\ Out$
shows $s1'\ x = s2'\ x$
$\langle proof \rangle$

end

References

[1] M. Eilers, T. Dardinier, and P. Müller. CommCSL: Proving information flow security for concurrent programs using abstract commutativity, 2022.

[2] V. Vafeiadis. Concurrent separation logic and operational semantics. In M. W. Mislove and J. Ouaknine, editors, *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS 2011, Pittsburgh, PA, USA, May 25-28, 2011*, volume 276 of *Electronic Notes in Theoretical Computer Science*, pages 335–351. Elsevier, 2011.